Exploring the Benefits and Implications of Dynamic Partial Reconfiguration using Field Programmable Gate Array-System on Chip Architectures

Beasley, Alexander

Award date:
2019

Awarding institution:
University of Bath

Exploring the Benefits and Implications of Dynamic Partial Reconfiguration using Field Programmable Gate Array - System on Chip Architectures

Alexander E. Beasley

A thesis submitted for the degree of Doctor of Philosophy

University of Bath

Department of Electrical and Electronic Engineering

December 17, 2018

Contents

List of Figures

List of Tables

List of Acronyms

2D 2-Dimensional

3D 3-Dimensional

AI Artificial Intelligence

ALM Adaptive Logic Module

ALU Arithmetic Logic Unit

ALUT Adaptive Look Up Table

ANSI-C American National Standards Institute for the C language

API Application Programming Interface

APM Automatic Parallel Module

APU Accelerated Processing Unit

ASIC Application Specific Integrated Circuit

BisFET Bilayer PseudoSpin Field Effect Transistor

BJT Bipolar Junction Transistor

BOX Buired Oxide Layer

CAD Computer Aided Design

CGRA Coarse Grain Reconfigurable Arrays

CMOS Complementary Metal Oxide Semiconductor

COTS Commercial off the Shelf

CPU Central Processing Unit

csv Comma Separated Values

DAB Digital Audio Broadcasting

DMA Direct Memory Access

DRAM Dynamic Random Access Memory

DSP Digital Signal Processor

DWARV Delftworkbench Automatic Reconfigurable VHDL generator

EMIF External Memory Interface

FeFET Ferroelectric Field Effect Transistor

FIFO First In, First Out

FinFET Fin Field Effect Transistor

FLOPS Floating Point Operations Per Second

FPGA Field Programmable Gate Array

FPGA-SoC Field Programmable Gate Array - System on Chip

FPL Feild Programmable Logic

fps frames per second

FPU Floating Point Units

GIT Global Information Tracker

GLSL Graphics Language Shader Language

GPP General Purpose Processor

GPU Graphics Processing Unit

HDL Hardware Description Language

HDR High Dynamic Range

HLL High Level Language

HPS Hard Processor System

I2C Inter-Integrated Circuit

I/O Input/Output

IC Integrated Circuit

IDE Integrated Development Environment

IP Intellectual Property

ISA Instruction Set Architecture

ISS International Space Station

LSB Least Significant Bit

LUTs Look Up Tables

MATLAB Matrix Laboratory

MGP Monochip Graphics Processor

MIPS Million Instructions Per Second

MISC Minimal Instruction Set Computer

MOS Metal Oxide Semiconductor

MOSFET Metal Oxide Semiconductor Field Effect Transistor

MPSoC Multi-Processor System on Chip

MSB Most Significant Bit

NHS National Health Services

NMOS Negative-channel Metal Oxide Semiconductor

OpenCL Open Computing Language

OpenGL Open Graphics Language

OS Operating System

PD Photo-Diode

PMOS Positive-channel Metal Oxide Semiconductor

PROM Programmable Read-Only Memory

PSK Phase Shift Keying

RAM Random Access Memory

RGB Red, Green, Blue

ROM Read Only Memory

SEU Single Event Upset

SDRAM Synchronous Dynamic Random Access Memory

SHA Secure Hash Algorithm

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Thread

SPI Serial Peripheral Interface

SoC System on Chip

SOI Silicon-on-Insulator

Tcl/Tk Tool Command Language/Tool Kit

TFET Tunnel Field Effect Transistors

TTL Transistor-Transistor Logic

ULP Unit of Least Precision

VCSEL Vertical-Cavity Surface Emitting Laser

VGA Video Graphics Array

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

VLIW Very Long Instruction Word

VLSI Very Large Scale Integrated

Abstract

Demands on modern computing are becoming more intensive. Keeping up with these demands has increasing complexity. Moore's Law is in decline. Increasing the number of cores on a device has diminishing returns. Specialised architectures provide more efficient and higher performing processors. However, it is not always practical to include every architecture on every device. Running non-native tasks on architectures often results in a drop in performance.

This research examines the benefits and limitations of Field Programmable Gate Arrays - Systems on Chip (FPGA-SoC) devices to provide flexible hardware accelerators for heterogeneous architectures. A number of topics are covered, including hardware acceleration of floating-point mathematical functions, dynamic reconfiguration and high-level synthesis. A number of case studies are presented. Dynamic reconfiguration is used to change the configuration of the FPGA at runtime, allowing the hardware accelerators to be changed depending on the current processor tasks. Changing accelerators at runtime has limitations, such as data perturbation. Context switching techniques are applied to the hardware to prevent loss of data and enable de-fragmentation of the FPGA. High level synthesis techniques are used in conjunction with the presented hardware accelerators to synthesise high-level languages into hardware descriptions with optimisations. Techniques for runtime synthesis of hardware accelerators are presented. These can be combined with dynamic reconfiguration to configure FPGAs with appropriate hardware accelerators from a high-level language at runtime.

The research demonstrates that FPGA-SoC devices have the potential for providing reconfigurable accelerators for processors in heterogeneous architectures. Metrics show that the FPGA configurations can perform better than other commercial processors. It was demonstrated that it is possible to context switch hardware at runtime, meaning the most can be made of the FPGA-SoC at all times, even as situations change. However, there are many limitations that still need to be overcome, such as management of the implemented hardware, synthesis of new hardware at runtime, reconfiguration times, interfacing of hardware with software and the design of hardware accelerators.

Chapter 1

Introduction

The world is becoming even more connected. The desire to do more with ever smaller devices is becoming greater. Data centres must provide faster service while searching larger databases. To keep pace with change, more inventive ways of constructing processors must be considered.

In 1965, Moore (co-founder of Intel) observed how transistor density had been increasing over the years and predicted this would continue. This later became known as Moore's Law. Although the exact wording has been forgotten, the general consensus is that Moore's Law states the density of transistors doubles every one to two years. The Moore's Law prediction has been correct and transistor density has vastly increased, as shown in Figure 1.1. Much

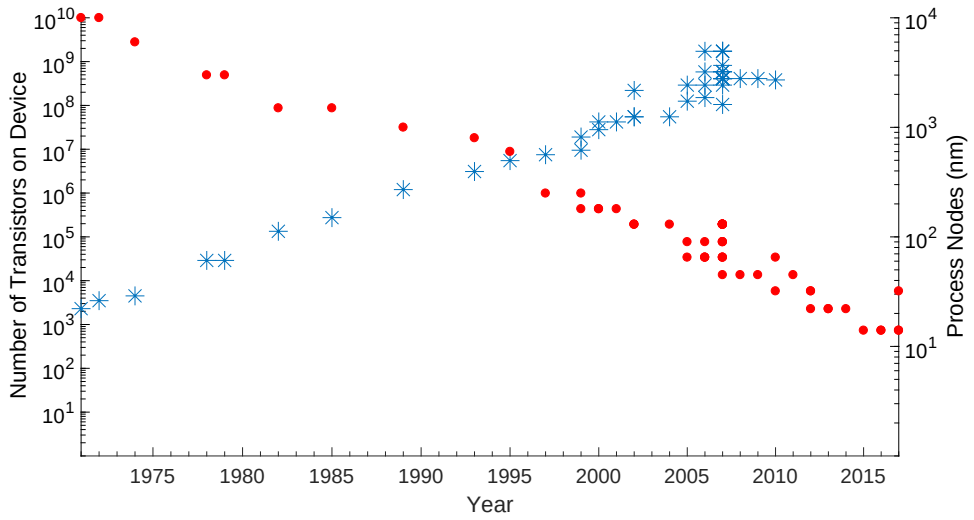


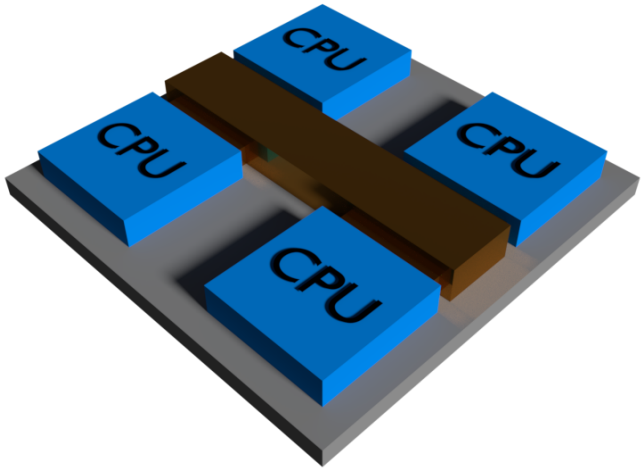
Figure 1.1: From 1970 to today there is a trend in the number of transistors in processors and the process nodes used in their fabrication. The data presented here has been collated from the Intel Ark of previous devices [4] and Intel history publications and articles [5] and [6]. Red dots are process nodes. Blue stars are number of transistors.

of this increase is due to improvement in process node technology, from 10 μm in 1971 to

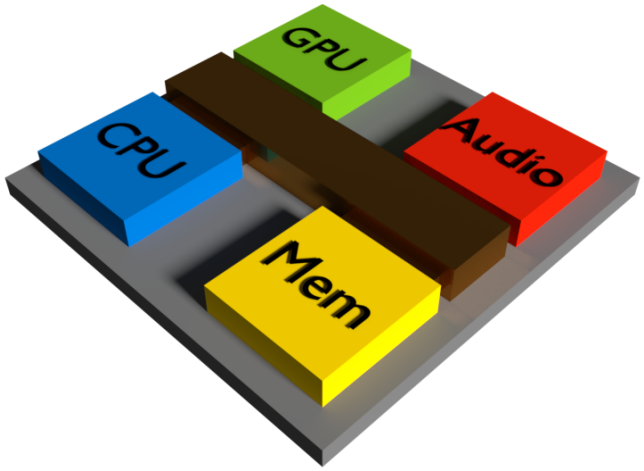
sub-10 nm today. It is predicted that this will reduce even further in coming years. However, smaller transistors present more challenges; for example, noise sensitivity becomes more of a problem.

'FinFET' technology, invented in 1999 by Hu [7] and first used commercially by Intel in 2011, radicalised how transistors could be designed. Hu first demonstrated this technology in the production of a sub-50 nm Positive-channel Metal Oxide Semiconductor (PMOS) transistor. The noise immunity of the FinFET design has lead to smaller, sub-10 nm, process nodes. Increasing transistor density, while important, can only improve processing capability so much. This can be seen as the limits of Moore's Law are reached. Instead, the use of load sharing and specialist architectures can greatly enhance the performance of a system.

1.1 Systems with multiple processors



(a) Homogeneous



(b) Heterogeneous

Figure 1.2: High-level representations of homogeneous and heterogeneous system architectures

Homogeneous architectures, shown in Figure 1.2a, use multiple processors of the same topology to share processing load, parallelise computation, and increase throughput. Examples range from small, single chip, multi-core, embedded devices to large server centres, such as those owned by Microsoft and Google, and all steps in between. Multi-core processing has been widely adopted due to its ability to increase throughput and separate tasks and it can be seen in all manner of applications. However, each processor is the same and can suffer from the same potential pitfalls.

Heterogeneous architectures, Figure 1.2b, combine different processors, exploiting the different specialisms of each topology for particular tasks. For example, mobile phones and laptop or desktop computers typically have a number of different processors. It is recognised that heterogeneous arrangements are one of the best ways to build a system to ensure the best performance. Many single-chip systems are now heterogeneous: consumer processors include on-board graphics in the same chip as the computational processor.

Even in a heterogeneous architecture there is no guarantee that there will be a specialised processor for all situations. Code must be written to match the target architecture. This may not always produce the most optimised results. Introducing a flexible architecture that can be adapted to the processing task could lead to a much more optimised system overall.

Field Programmable Gate Arrays (FPGAs) provide a flexible device on which specialised architectures can be implemented. System on Chip (SoC) devices combine GPPs, such as the ARM Cortex-A range, with FPGA fabric in the same package. In 2015, [8], Intel acquired Altera, one of the largest FPGA vendors. This merger is likely to produce more FPGA-SoC devices.

1.2 Current processing technologies

1.2.1 General Purpose Processor - *GPP*

A GPP is a processor that can perform a variety of tasks without being overly specialised in one. GPPs are often the basis of a system where control-flow, such as '*if-else*', statements are dominant. Other architectures, such as graphics processors or FPGAs, are better suited to data-flow applications where deep pipelines and high levels of parallelism can be implemented.

The most commonly recognised form of GPP is the Central Processing Unit (CPU) used in every computer. The CPU's ability to perform all tasks makes it suited to managing a system. However it may not be the most efficient at certain tasks, such as floating-point

operations using graphics rendering. A CPU, shown in Figure 1.3, is good at managing the interactions between various different hardware, e.g., GPUs and Random Access Memory (RAM), and performing Operating System (OS) instructions.

Modern CPUs are characterised by their large number of cores and multilevel memory caches. In 2017 two of the best-known processor manufactures, Intel and AMD, announced new devices with ten or more processor cores, each of which supports hyperthreading, [9, 10, 11]. A typical processors can have three levels of local cache (L1, L2, L3). Typically L1 caches are kilobytes, L2 caches are hundreds of kilobytes and L3 caches are several megabytes. Caches provide a small amount of memory physically close to the processor's Arithmetic Logic Unit (ALU) to reduce latency thus increasing throughput. Lower levels of cache (L1 and L2) could be specific to a single core, but are smaller. A L3 cache could be shared between multiple cores on a device, Figure 1.3. However caches are small so it is not guaranteed that data will be there. Hence processors are nondeterministic devices, meaning the time taken for an operation cannot be predicted accurately and is subject to change.

A typical architecture for a processor is given in Figure 1.3. Each manufacturer (e.g. AMD, Intel, IBM) has architectural differences that make their products unique.

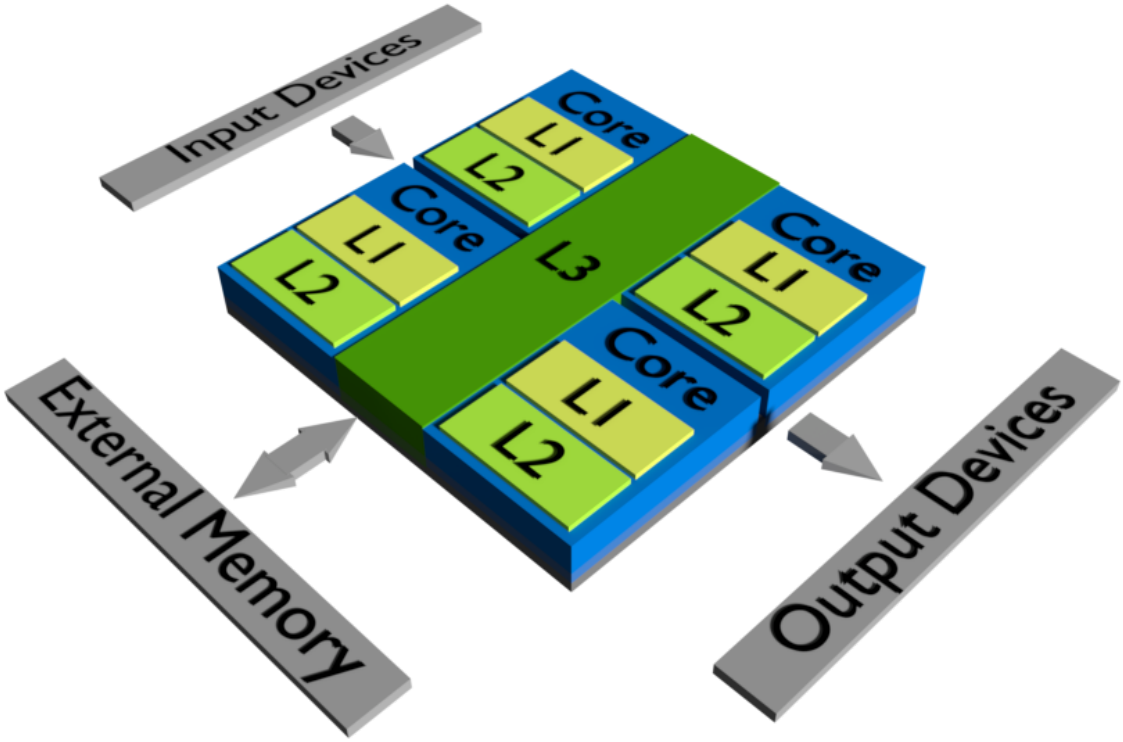


Figure 1.3: Overview of a typical CPU, reference from Intel [1].

1.2.2 Graphics Processing Unit - *GPU*

GPUs are designed primarily for intensive graphics rendering tasks, which have floating-point mathematical operations. However, more uses of GPUs are being recognised such as data mining (Bitcoin [12] and Ethereum [13]), neural network and artificial intelligence, and Compute Unified Device Architecture (CUDA) [14]/Open Computing Language (OpenCL) [15] applications.

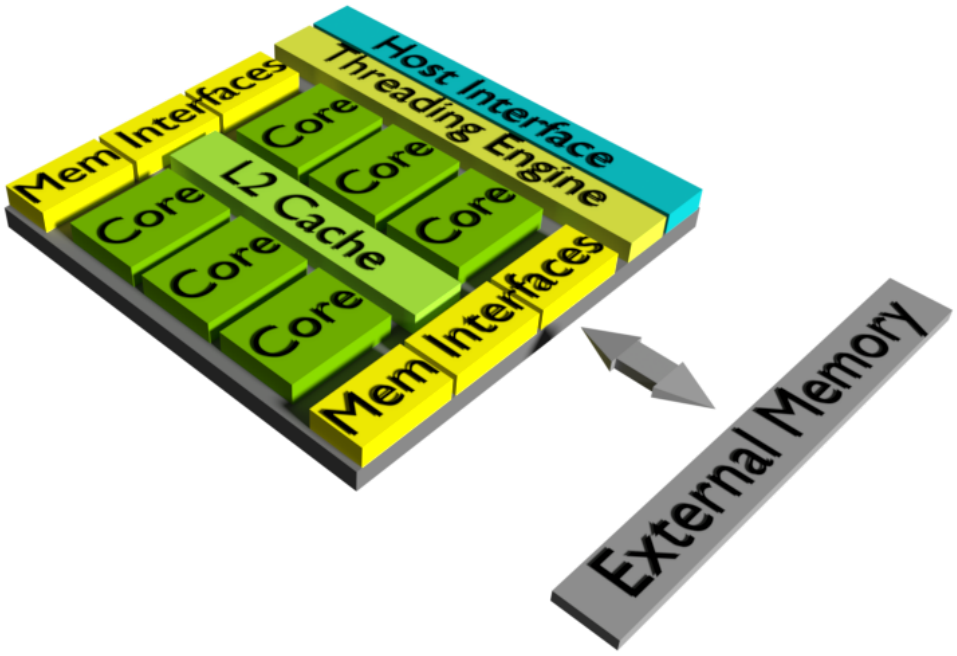


Figure 1.4: Overview of a typical GPU, reference from NVIDIA [2].

GPUs are designed for higher data throughput that GPPs. They have a large number of cores which have several main functional blocks optimised for processes key to graphics rendering, Figure 1.4. The high core count allows the application to be written to exploit parallelism. Threading separates the data stream to run processes in parallel. GPUs typically have a dedicated threading engine. Data pipelining increases the overall throughput of the device by minimising the number of operations that must be executed at each stage. A data pipeline is a set of processing elements connected in series with each other. Generally, a deeper data pipeline yields higher performance. A high-speed interface allows fast communications between the host and the GPU. A large number of high-speed memory interfaces, coupled with an L2 cache, ensure low latency transfer of data to each of the processing cores. Graphics cards often use the Single Instruction, Multiple Data (SIMD) paradigm [16]. This allows the multiple processing cores to perform the same operation on multiple data points, increasing the throughput, so long as data alignment is maintained. Standard programming paradigms

for GPUs and SIMD architectures target vectorised code. However, not all code can be vectorised, such as control-flow intensive processes like parsing.

This optimisation results in trade-offs. GPUs are good at high-speed mathematical computation on vectorised data or parallelisable problems. However, GPUs are rigid in the way they process data. Hence, compromises must be made to fit the task to the architecture.

1.2.3 Accelerated Processing Unit - *APU*

APUs use a simple heterogeneous architecture, combining a CPU and GPU in a single device. APUs achieve a high computational throughput and reduce the overall processor footprint and design complexity. The CPU is used to handle the everyday running of a system, while computationally intensive tasks can be offloaded to the GPU.

APUs can also include video codecs for the graphics card, and memory controllers to allow different elements to communicate. These devices are suited to space and power constrained situations. However, they will still suffer from the limitations found in both the CPU and GPU individually.

1.2.4 Digital Signal Processor - *DSP*

DSPs are designed for high-speed data processing of real-time signals with highly optimised Input/Output (I/O). The type of calculations performed on a DSP often require floating-point operations. DSPs use pipelined multiply-accumulate blocks with hardware controlled looping to achieve minimal - sometimes zero - overhead. A DSP maybe used as a slave device for high-speed data processing off of the systems processor.

DSPs typically contain local RAM, Read-Only Memory (ROM), and an instruction cache along with external memory interfaces. Much like caches on processors, local RAM and ROM store data close to the core to reduce latency. As DSPs are often used in embedded systems, they tend to contain interfaces, such as Inter-Integrated Circuit (I^2C) and Serial Peripheral Interface (SPI), for communicating with external devices.

1.2.5 Field Programmable Gate Array - *FPGA*

Unlike the devices listed above, FPGAs are configurable hardware, not processors. The configurable nature of an FPGA allows the designer complete customisation of the data types. FPGAs are used for re-routable PCB design, hardware acceleration, and are also powerful tools for the development and verification of Application Specific Integrated Circuits (ASIC).

This will be further addressed in Section 1.2.6.

FPGAs are versatile and have a variety of integrated Intellectual Property (IP) such as memory, DSPs and transceivers. Some resources; e.g. memory and DSP blocks (Figure 1.5), are interlaced with the FPGA logic fabric for ease of access and to minimise routing delay. Unlike discrete DSP devices, DSP blocks in an FPGA are a multiply-accumulate stage. In a modern FPGA, the DSP resources are becoming more capable and can be configured to a number of modes such as high-, standard-, or floating-point precision [17]. Other resources, such as External Memory Interfaces (EMIF) and transceivers, are found on the edge of the chip as they are concerned with I/O applications.

When programming for traditional processors, the task is tailoring a problem/algorithm to a given architecture. When programming for an FPGA, the task is somehow the other way round by tailoring the architecture to the problem in the first place. FPGA-SoC devices are becoming more prevalent. These contain all the standard features found in an FPGA, along with a Hard Processor System (HPS). The HPS is a GPP that shares the same silicon as the FPGA fabric. The interface between the two devices can then be detailed by the design engineer resulting in a custom, flexible, fast interface.

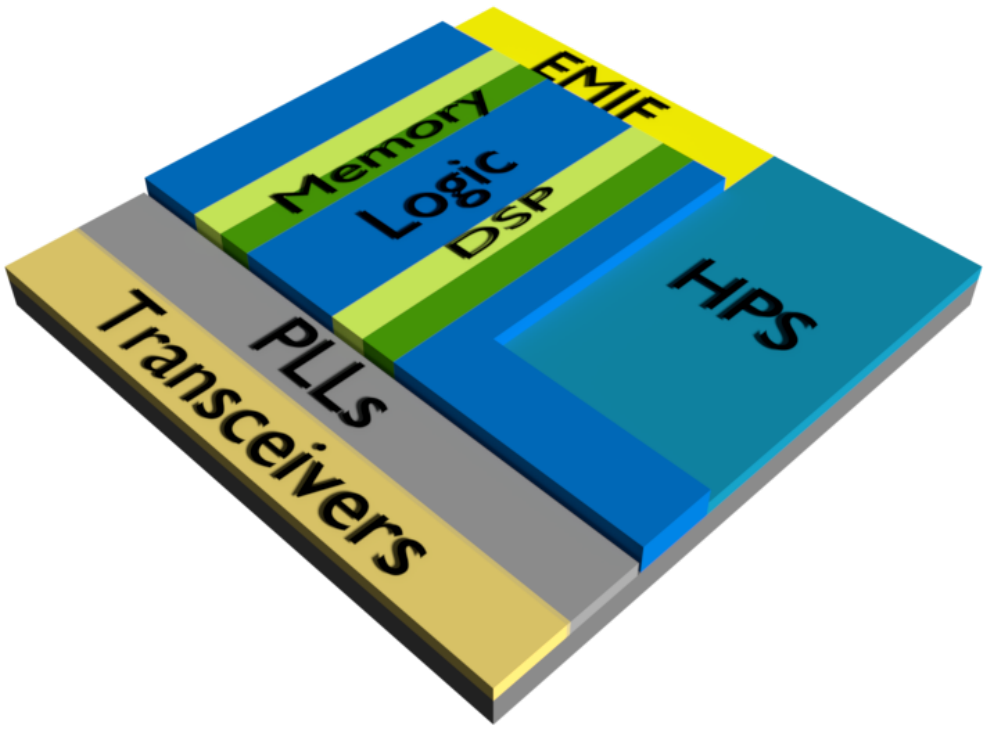


Figure 1.5: Overview of a typical FPGA-SoC, reference from Intel FPGA [3].

1.2.6 Application Specific Integrated Circuit - *ASIC*

ASICs are designed to perform a single task repeatedly. This specialisation leads to a very optimised but very inflexible solution.

An ASIC's architecture is dependent on the environment and task for which it is developed. The details of many devices are considered company secrets. An ASIC could be a specialised processor with a limited instruction set or a single hardware accelerator.

No two tasks are necessarily the same and neither are any two processors. There is also variation within a single type of processor. Products aimed at consumers must be as versatile as possible to meet the markets demands. Application specific offerings, which are much more commonplace in aerospace and military sectors, must focus on reliability.

1.3 Aims and Organisation of this Thesis

The work presented in this Thesis is based on Intel FPGA (formally Altera) technology. Most current research focuses on Xilinx Technology, whereas a key feature of this research is the use of Intel FPGA devices. This thesis explores a number of aspects key to using Intel FPGAs for dynamic reconfiguration. There are many implementations of floating-point mathematical functions available as Intellectual Property (IP). However the first part of this research designed new cores specifically for use with the applications presented later. The use cases are Graphics rendering on an FPGA and creating an implementation of the Hodgkin-Huxley neuron spiking model. By not using pre-existing library implementations of these functions the 'black box' aspect is removed; the implementation is more transparent and would allow for greater control over tuning for certain performance metrics.

The next part of the research focused on implementing partially reconfigurable architectures on Intel FPGA devices. The research considers FPGA architectures for both end-to-end data flow and hardware acceleration of processor based tasks. The context switching of FPGA-based hardware accelerators is presented, discussed and demonstrated.

Finally, the research considers high-level synthesis tools for FPGA design. The research presents the design for a high-level tool that is designed to convert from Graphics Language Shader Language (GLSL) code into an FPGA implementation using the library of floating-point functions developed for this research is presented.

The work detailed within this PhD Thesis addresses the following questions:

(Q1) What is the state-of-the-art for FPGA accelerated processes, dynamic reconfiguration

and high-level synthesis?

(Q2) What are the performance benefits of heterogeneous system architectures?

(Q3) How do these compare to traditional single and multiple processor topologies?

(Q4) What are the performance benefits of dynamically reconfigurable FPGA resources?

(Q5) What are the requirements and limitations of heterogeneous architectures on FPGAs?

Chapter 2 reviews and critiques current literature, specifically hardware acceleration, dynamic reconfigurability, high level synthesis, and context switching (Q1). Chapters 3 and 4 demonstrate the use of FPGAs as accelerators of floating-point functions, specifically computationally expensive functions such as add, multiply, vector operations, reciprocal, square-root, and exponential (Q2). Chapter 4 implements the Hodgkin-Huxley model of a neuron using the FPGA, capable of processing at a rate of $100\,s.s^{-1}$ (Q3). Chapter 5 provides a case study for creating a GPU on the FPGA using floating-point operations, comparing power requirements and throughput with Commercial Off-The-Shelf (COTS) devices (Q2 and Q3). Chapter 6 details how dynamic task allocation and context switching can be implemented on an FPGA, allowing hardware accelerators to be changed at runtime (Q4 and Q5). Chapter 7 provides the design and methodologies for synthesis of HDL from a high level language (Q5). Finally, Chapter 8 concludes the Thesis, summarising the main results and outlining further works.

The novel contributions of this research are: resource optimised FPGA implementations of the square-root and exponential functions (Chapter 4); an implementation of the Hodgkin-Huxley neuron on an FPGA using linear piecewise and curve-fitting methods (Chapter 4); the implementation of a GPU using only FPGA fabric (Chapter 5); a new type of flip-flop used to create context switchable FPGA designs with a method to automatically convert the gate level verilog into a context switchable design (Chapter 6) and a high-level synthesis tool for GLSL code (Chapter 7).

Chapter 2

Review of Literature

The research topic covers a number of distinct areas:

- Processor topologies, their development, and their potential decline
- System architecture designs
- Hardware acceleration
- Selecting and scheduling hardware tasks
- Switching between tasks
- Techniques for the automatic generation and optimisation of hardware designs

These individual areas have been the subject of research for many years. However, this does not mean their research potential has been exhausted. Before conducting further research in this area, it is important to understand the literature in each area and how they cross over. This Chapter will assess a number of previous works, discussing the impact they have on this work and providing the background and motivation.

2.1 Processor topologies

This research will explore the use of flexible Field Programmable Gate Array (FPGA) fabric to replace traditional processor topologies and enhance heterogeneous architectures. It is logical to first consider processor topologies and how they have become part of everyday life. An introduction to the various different types of processors has been given in Chapter 1. The development of these different processors shall now be considered in more detail.

2.1.1 The early years

2.1.1.1 Moore's Law

Perhaps the most famous, albeit misquoted, 'law' for electronics is attributed to Moore. In 1965 Moore wrote a prediction for the 35^{th} anniversary edition of Electronics Magazine regarding the course of the electronics industry over the coming ten years. Moore discussed the economic standings of the semiconductor industry and how the reliability of integrated packages made them appealing for new ventures, such as space missions and in the military [18]. Evaluating the costs and yields of semiconductor fabrication, Moore predicted that "by 1975, the number of components per integrated circuit for minimum cost will be 65,000". Moore also stated that the "principle advantages will be lower cost and greatly simplified design-payoffs from a ready supply of low-cost functional packages". Moore's article secured his place in history by considering the importance of silicon to the semiconductor industry.

Since then, Moore has appeared as a keynote speaker on multiple occasions, discussing the development of the semiconductor industry and the potential roads it must explore. In 1979 Moore attended the International Solid-State Circuits Conference where he posed the question "Are we really ready for [Very Large Scale Integration] VLSI[2]?". Microprocessors were already well established and had been seen as a "cost-effective solution to a broad range of applications". However, the next step concerned how hardware could be used to maintain "system advantage" with hardware acceleration [19]. Almost four decades ago the importance of custom hardware and how it interplayed with flexible processor technology was being considered.

In 1995 Moore wrote an article redressing what had become known as Moore's Law [20]. He acknowledged that Moore's Law had become associated with "almost anything related to the semiconductor industry that when plotted on semi-log paper approximates a straight line". The article compared the predicted ten-year growth of transistor density between 1965 and 1975 with what actually occurred over the ten year period. It was found that Moore's Law had held relatively true and the semiconductor industry had grown at a remarkable pace. Digital computing was widely available and the demand for higher performance was being met. The rapid growth of the industry was not necessarily countering the growing costs it was facing. Yield increases were beginning to plateau and newer ways were needed to improve value.

Moore presented the significant growth in the market of Intel branded processors at the 2003 International Solid-State Circuits Conference [21]. Moore focused on the economic

growth of the industry and states that the past 30 years had seen an average compound annual growth of 15%. However, the number of transistors that had been produced grew by 78% on average with several years in the '70s and '80s experiencing more than 100% growth. This phenomenal increase had been accompanied by a reduction in size and increase in efficiency of transistors. The cost of manufacture had significantly reduced and minimum feature sizes were orders of magnitude smaller than in 1965. Although the semiconductor industry had enjoyed the unprecedented growth, challenges were beginning to emerge that could see an early end to the now critical industry. New 'Tri-gate' structures, which radically change traditional transistor design were being developed. They will be discussed later in Section 2.1.

There are a number of limitations and challenges facing the Moore's Law prediction. Over the years different approaches have been used to overcome a variety of challenges; it is these that will be discussed.

2.1.2 Specific hardware for specific tasks

It is well established that, just like no two tasks are necessarily the same, neither should the approach for solving them be. While it is true that a General Purpose Processor (GPP) is capable of almost any task, its efficiency can vary massively and is often very low. An algorithm or process may require a higher number of calculations, which also increases time taken and power consumed. As demands for both high throughput and low power consumption increase, introducing specialised hardware designed for specific tasks is a must.

An example of specialised hardware is the Graphics Processing Unit (GPU). These are now commonplace in every day life. In 1980 the concept of a specific processor for graphics was introduced [22]. The 'true graphic processor' provided dedicated hardware designed to interface with any 8-bit microprocessor. Branded as a 'high performance' processor for the time, it managed to display a 512×512 portion of a 4096×4096 pixel image with a raw speed of 2 MPixels/second . This type of technology has lead to a whole new market sector, today worth billions of dollars.

The development of specialist architectures progressed rapidly. Some new processors even became more application specific than generic graphics processors. In 1982 Liu & Eastman wrote about a GPU for computer-aided drafting [23] and in 1983 Nishimura *et. al.* produced a colour graphics processor for television broadcasting [24]. These technologies focussed more on the user experience, removing the need for technical understanding to be able to make

full use of the devices.

The next step was to reduce the complexity of implementing specialist devices. In 1990, Geneste & Auger published work that unified multiple custom and semi-custom Application Specific Integrated Circuits (ASICs) for graphics processing onto a single Monochip Graphics Processor (MGP) [25]. Their publication summarised over a decade of graphics processing ASIC development, combining a total of nine ASICs and three Programmable Read-Only Memories (PROMs) into a single chip solution. The resulting MGP was fabricated using a 1 μm Complementary Metal Oxide Semiconductor (CMOS) process.

Graphics processing is not the only type of task that benefits from application-specific hardware. Often digitised data must be processed at high speeds to represent analogue waveforms. This task is easily carried out by a Digital Signal Processor (DSP). Uses for DSPs can be varied, such as neural networks, due to their optimised Input/Output (I/O). It is hoped that creating a system that operates by mimicking the brain and learns will produce vast amounts of computing power. This is an incredibly difficult challenge. In 1989 Suzuki & Atlas discussed how analogue VLSI may produce an optimal processor for neural networks. Given that replicating mathematics in a VLSI environment can be difficult, they proposed "a digital implementation as a reasonable near-term alternative" [26]. Pipelining techniques are mentioned to increase throughput. It is now widely accepted that pipelining is important to achieve very high data throughput.

Transistors have become smaller in a very short space of time, but the smaller they become, the harder it is to make them smaller. The next Section will look at the development of CMOS technologies and possible new directions.

2.1.3 The history of CMOS

Digital devices use a large number of discrete semiconductor devices on one chip to create functions. Silicon is doped to have an excess (n-type) or lack (p-type) of electrons, which allows electron flow under specific conditions. Diodes provide the simplest semiconductor device consisting of a single p-n junction. Some basic logic functions such as AND and OR are possible with 'diode logic'. The logic inputs are used to control the bias of diodes to set the output. However, the introduction of transistors provided a considerably better and more flexible way to construct discrete logic gates. Transistors allowed the invention of Transistor-Transistor Logic (TTL), followed by CMOS logic. This progressed with more transistors being included in single packages, eventually creating processors as they are known today.

TTL is constructed from Bipolar Junction Transistors (BJTs). The logic inputs are used to drive current into the emitter of a BJT. Current is forced to flow from the base to the collector of the transistor. This controls the base on a second BJT to set the output. A NAND and a NOR gate constructed using TTL are shown in Figure 2.1.

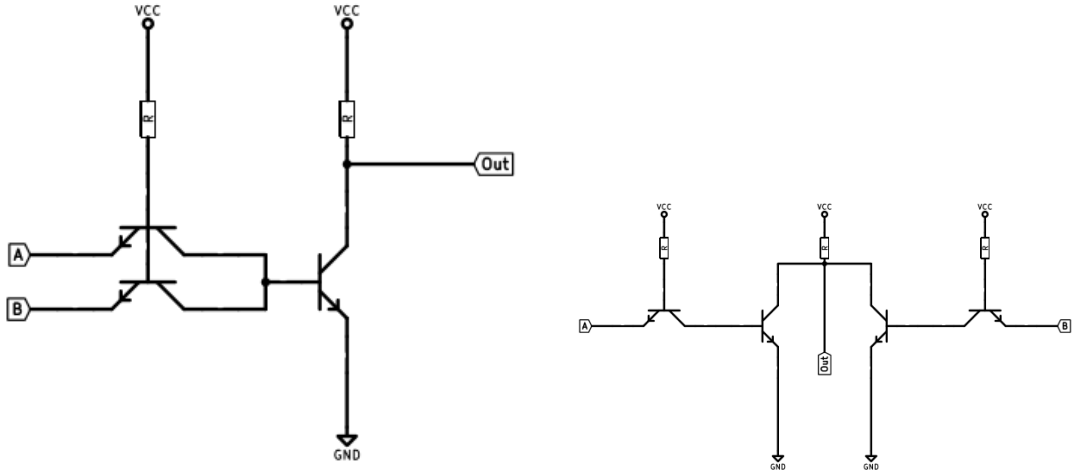
The truth tables for NAND and NOR gates are shown in Tables 2.1 and 2.2 respectively.

Table 2.1: NAND gate truth table

A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

Table 2.2: NOR gate truth table

A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0



(a) NAND gate

(b) NOR gate

Figure 2.1: TTL implementations of basic logic functions

Complementary Metal-Oxide-Semiconductor (CMOS) technology, show in Figure 2.2, was introduced in 1963 by Sah & Wanlass of Fairchild [27, 28]. CMOS moves away from current-driven BJT transistors to voltage-controlled Metal Oxide Semiconductor (MOS) transistors. CMOS technology is a combination of both P-channel and N-channel Metal Oxide Semiconductor, Field Effect Transistors (MOSFETs); every P-channel has a corresponding N-channel (and vice versa). The are arranged so that every nMOSFET is supplied by either V_{SS} or another nMOSFET and every pMOSFET is supplied by either V_{DD} or another pMOSFET. The result is a very low power consumption device which generates minimal heat. However, they can struggle to operate at high switching frequencies. A NAND and a NOR gate using CMOS technology is shown in Figure 2.3.

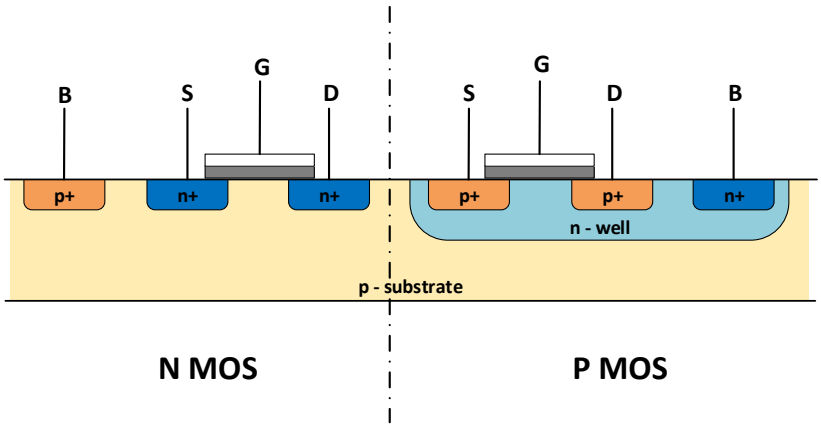
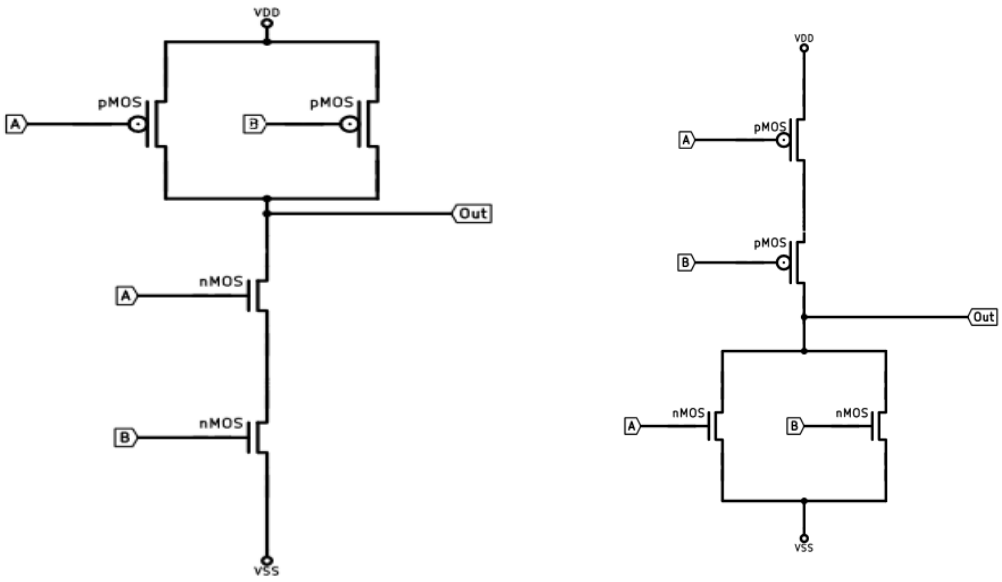


Figure 2.2: CMOS circuits combine both n- and p-doped transistors on the same wafer. B is the bias, S is the source, G is the gate, D is the drain, p+ is positively doped silicon, and n- is negatively doped silicon.



(a) NAND gate (b) NOR gate

Figure 2.3: CMOS implementations of basic logic functions

The process node is defined as the minimum feature width. CMOS technology has allowed the rapid reduction in process node technology: smaller processes lead to more densely packed transistors with higher performance and lower total power dissipation per operation. This allowed the complexity of general purpose and application specific processors to increase. However, conventional CMOS technology has its limitations. Scaling the size of a MOSFET to below 0.5 V becomes difficult due to its 60 mV per decade swing of current [29]. As a result, there are many other technologies that are being investigated to increase performance: Tunnel Field Effect Transistors (TFET) [30, 31]; Ferroelectric Field Effect Transistor (FeFET) [32, 33]; magnetic spin and orbital state Bilayer PseudoSpin Field Effect Transistors

(BisFETs) [34, 35]. Additionally other materials such as graphene and carbon nanotubes are also being considered. Currently these technologies are still in their early experimental stages. One adaptation of the FET that is already used is the FinFET.

2.1.4 The FinFET

It has been shown that processor technology has seen dramatic change over the past 50 years. Process nodes have shrunk rapidly: in 1971 a typical process node would have been 10 μm, whereas nodes today are sub-10 nm. This dramatic reduction in feature size was assisted by Hu *et. al.* in 1999. They proposed a new arrangement for the standard transistor design, repositioning the gate of a transistor, resulting in the 'FinFET' [7, 36], Figure 2.4.

The intention of the FinFET, also known as the 'Tri-gate' structure, was to create sub-50 nm PMOS transistors. Fabricated prototypes showed good performance characteristics down to 18 nm. Even when the research was first published, simulations showed promise of a sub-10 nm minimum feature size. Only recently have such small process nodes started to be used in production. Despite early praise from Moore at the International Solid States Conference in 2003 [21], the FinFET did not come into use until about a decade later.

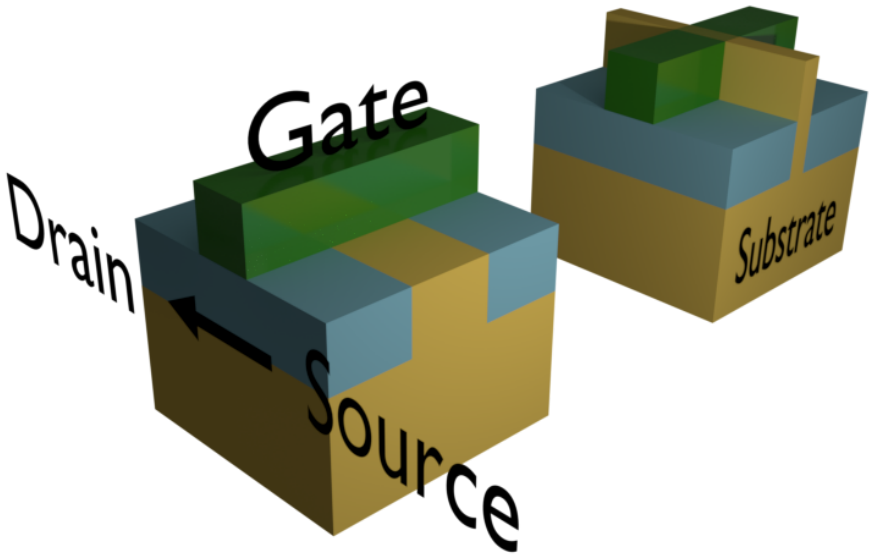


Figure 2.4: Compared to a regular transistor, the FinFET (right) repositions the source-drain structure to create a 'fin' shape allowing fully depleted operation. Traditional MOSFET construction (left) has the gate on top of the source-drain substrate. The gate is shown in green, substrate in yellow, and oxide layers are shown in blue.

2.1.5 The photonic processor

It is not possible to continually shrink transistors as fundamental physical limits will eventually be reached.. Eventually it will not matter how small a transistor can be made, it will not be possible to overcome noise effects and other limitations will dominate a systems overall performance. Whilst smaller transistors mean higher transistor density and shorter tracks, it still takes time to move charge from one location to another. Although signal propagation in a copper wire can be faster than in optical fibres, electrical interconnectivity is slow when compared to the speed of light. Depending on the application, this property can be exploited.

Research has been conducted into producing photonic-inspired CMOS devices [37, 38]. Initial findings indicated an increase in energy efficiency and throughput compared to electrically connected devices by up to ten times. In [37], Strojanovic *et. al.* reported a four times energy efficiency and throughput gain for core-to-core performance and a ten fold improvement for core-to-Dynamic Random Access Memory (DRAM) networks.

Shen *et. al.* presented the designs for a Silicon-on-Insulator (SOI) 3D guided-wave path device in [39], shown in Figure 2.5. The performance was reported to be 10 Gbps error-free data transmission using 9 mA driving current. The promise of such high throughput devices has inspired other related research. Yang *et. al.* presented an 'Optical Matrix Processor' in 2016 [40] that performed 1.6×10^9 multiplication and accumulation operations per second. Further, Vinckier *et. al.* demonstrated how photonic processing can be used to process high bandwidth signals for low power through use of a neural network application [41]. Such devices appear to be a promising solution to the inevitable end to Moore's Law and the lightning-fast development of the semiconductor industry. However, they do not appear commercially viable in the near future.

2.1.6 What is the next step?

Embedded technologies have developed rapidly since their invention. Process nodes have shrunk, yields and efficiency have increased. The necessity of application specific devices such as the GPU and DSP has been confirmed and they are now ubiquitous in modern society. However, as process technology begins to throttle Moore's Law and no alternative is available yet, a different approach must be found.

One obvious way forward is unifying various architectures into single-chip solutions. This would allow better use of the specialised architectures and computational load sharing to increase efficiency and throughput.

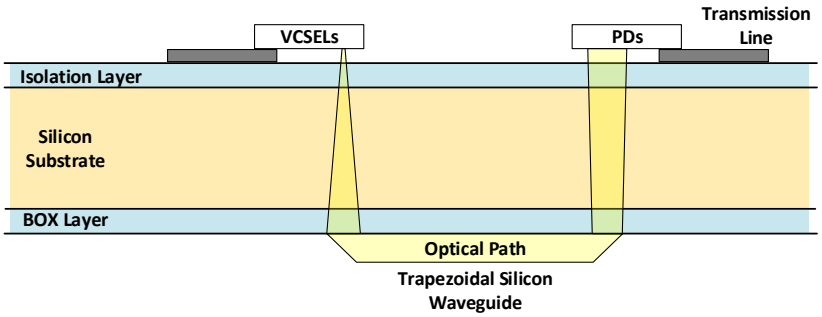


Figure 2.5: Photonic processors introduce waveguides so that photons can be used in place of electrons to trigger FETs. Image drawn using [39] as a reference. Where VCSEL is a Vertical-Cavity Surface Emitting Laser, PD is a Photo-Diode, and the BOX Layer is an SiO_2 Buried Oxide Layer.

2.2 Architectures

2.2.1 What makes the 'best' architecture?

Different situations have different requirements. A system could require low power, small size, high throughput or efficiency. Hence, the architecture could be tailored to give the best performance for a given situation. Lessons learnt from early architectures for devices and systems are key in their continued development.

The interaction between different components plays just as big a role in determining the overall system performance as the individual elements.

2.2.2 Homogeneity and heterogeneity

When building a system, the architecture will fall into one of two main categories: homogeneous or heterogeneous. These have already been discussed in Chapter 1. It is important to understand the differences between the two concepts and the advantages and limitations of both. This Section will discuss the arrangements of different systems within both categories and some potential uses.

Any system can be considered either homogeneous or heterogeneous. Which category it falls into is a matter of scope. For example, consider a desktop computer. Within a computer, there are a variety of different components, such as CPU, GPU and memory. These different items are connected to form a system which is heterogeneous. Alternatively the system can be examined on a smaller scale. The processor may be made from multiple cores. If all of these cores are the same, it is a homogeneous system. Similarly the GPU is likely to be a

homogeneous system made of many identical elements.

On a larger scale, multiple identical computers could be connected to form a data centre. All of the computers are identical so a homogeneous system has been built. Therefore, there is now a homogeneous system made from heterogeneous systems made from homogeneous components.

The question still remains, however, which is better? Heterogeneous systems have a wider variety of specialisations. While it is true that a heterogeneous system will have better performance and energy efficiency [42, 43, 44, 45] this is only achieved with careful design.

In heterogeneous systems, the problem of synchronisation and communication becomes a limiting factor. Homogeneous systems naturally require high synchronisation for low energy consumption [43]. However, a homogeneous system cannot exploit the variety of different architectures of a heterogeneous system.

2.2.3 How multiple devices interact

Communication and synchronisation between various elements is key for system performance. To illustrate the impact, the example of a computer will be revisited. Each of the elements - CPU, GPU and memory - are discrete on the motherboard. They are controlled from the CPU, which passes data to the other system elements. This transfer of information is not instantaneous; it takes time and hence reduces overall system performance. In a computer the data and instructions must leave the source and travel across the motherboard, using a standard protocol, to the destination. If the two devices are the same, they would be synchronised and communication would be faster. However, a computer is heterogeneous, making synchronisation a problem.

While the computer is a relatable example, modern technologies focus on single silicon devices. Yousif *et. al.* presents communication considerations for heterogeneous and homogeneous systems [43]. Systems were categorised as: *Master-Slave*, *Pipeline* and *Net Architecture*. The *Master-Slave* system used a single processor to control the actions of the other processors, distributing tasks as necessary. The *Pipeline* architecture used multiple processors to provide pipelines for the other processors. This reduces the computation load but requires specific time control. The *Net Architecture* allowed all processors to talk to any other processor as there was no hierarchy.

Yousif *et. al.* also presented a variety of different intercommunication and memory access topologies. Data transfer between processors is important for performance. *Point-to-*

point transmission provided a direct communication link and there was no need to wait for other communications to finish. However, in large systems this is inefficient. A *shared bus* approach was a more efficient use of resources, but sharing the communication link between all processors requires time division sharing and performance suffered as a result. The *net-on-chip* approach is becoming more popular. It attempts to mitigate the shortcomings of the other two approaches, creating the best solution in terms of both performance and resource use.

Memory was also addressed by Yousif *et. al.* [43]. Fast processor communication is important, but memory access is equally vital. Yousif *et. al.* discussed *shared memory* and a *message passing technique* at length. Shared memory is accessible via all the processors. This requires arbitration to avoid conflicts which can slow a system down. Message passing techniques distribute memory around the system so that each processor has its own local memory. Yousif *et. al.* concluded that *net-on-chip* and *message passing* techniques were the best approaches for fast communications between system elements.

The arrangement and shape of the elements is also important. Xiao & Baas conducted research into arrangements of different n-sided elements for greater communications rates [46, 42]. A processor die was considered as a four-sided element in contact with another four elements in a 2D mesh. Xiao & Baas experimented with offsetting the processor elements, similar to the construction of a wall, to increase the number of neighbours. They also considered a range of other tessellating polygons to see if performance gain could be achieved. It was measured using standard benchmarks. It was reported that six-neighbour hexagon tiles performed better than a four-neighbour mesh. However, each tile increased the fabrication area by 2.9%. Overall the increase in interconnectivity performance meant the total area was decreased by 22% with a average power saving of 17%.

Transistor process nodes have become smaller, to the point where they now provide excellent throughput for very low energy cost. Hence communications between processors is starting to dominate costs. Communication protocols and interconnectivity must be designed carefully for the most efficient use of a collection of processor technologies.

2.2.3.1 Multi-cored single chips

Modern processors contain multiple identical cores. Increasing the number of cores improves throughput and power efficiency. Development of homogeneous processor architectures shall be discussed in more detail.

Early multi-cored processors consisted of a homogeneous arrangement of processing cores, with an increasing number of cores to provide more computational power. Yousif *et. al.* considered the impact of communication and memory arrangements. This study came after the initial development of multi-core processors.

A homogeneous system provides fewer advantages than a heterogeneous system. The cache implementation in a homogeneous processor is an important factor for performance. Nakajima *et. al.* compared two cache implementations for a dual-core homogeneous processor: *dual-port* and *snoop access*. Arranging the cache to allow multiple accesses for minimal overhead gives large performance advantages [47]. Results showed the dual-port method reduces the power dissipation by 23% and requires 29% less floor space. Memory arrangements for heterogeneous architectures are equally important.

Soleymanpour *et. al.* demonstrated a 42.71% speed increase and an energy reduction of 30.77% for *network-on-chip* communications when compared to the homogeneous counterparts [44]. However, while 'cutting and pasting' works for a homogeneous processor, every part of a heterogeneous system must be individually, and carefully, designed to ensure synchronisation. Soleymanpour *et. al.* proposed a synthesis algorithm based on a commercial design tool for a Million Instructions Per Second (MIPS) processor. The algorithm extracted custom instructions and explored the design space to implement a system with optimal throughput and power consumption.

Sarma & Dutt discussed cross-layer heterogeneous architectures in [48]. They considered several different design aspects including application, Operating System (OS) and hardware, balancing them to provide potential Multi-Purpose Processor on Chip (MPSoC) configurations. Given the number of variables in a system, providing a completely optimised solution is difficult. The work in this paper considered only single instruction set architectures, for simplicity. The system used a predictive cross-layer approach which selected the most promising outcome. It did provide a large decrease in simulation time, but still required the input of a designer at the final stage.

Souza *et. al.* noted that although MPSoCs have allowed for rapid development in the embedded world, they lack cross-platform software compatibility [49]. There could be as many different instruction sets as there are MPSoCs, therefore each program must be rewritten to match its target. In contrast, GPPs tend to share a common instruction set, greatly decreasing development time. Souza *et. al.* proposed a heterogeneous multi-core with a homogeneous instruction set using a "binary translation mechanism". The intention was to

recompile the code to match the target at runtime.

In all architectures the energy efficiency must be considered. Processors under load get hot and the heat reduces a system's performance, or even stops it working altogether. In 2010 Ge & Qiu studied the effects of task allocation on the power consumption of a homogeneous multi-cored processor [50]. The study considered the leakage current and the fan power, which are vitally important in ensuring the system runs optimally.

2.2.3.2 Heterogeneity on different architectures

GPUs take advantage of thread level parallelism to distribute the incoming data and achieve high throughputs. This translates to limited capability for instruction level parallelism [51]. Xiang *et. al.* proposed a heterogeneous graphics processing architecture constructed of cores either capable of thread level parallelism or instruction level parallelism. The heterogeneous architecture achieved higher throughput, energy and area efficiency than a homogeneous version that used only one of the two available parallelisms.

Work that exploited a heterogeneous multi-core system with GPU acceleration is presented in [52]. Particle swarm optimisation, which is inherently computationally expensive, was considered. The task would not run efficiently on a single-cored processor. Wachowiak *et. al.* used a system that comprised multi-cored processors, graphical acceleration and Intel Xeon Phi co-processor units (with vectorisation). The results showed that large speedups are possible. The authors suggested that a heterogeneous approach could be used on other time-intensive stochastic problems.

No matter the technology, there are a variety of different architectures that could be exploited. Generally a heterogeneous system will outperform counterparts that rely on a single technology. However, the design of heterogeneous architectures poses its own problems. It is possible to apply lessons learnt form homogeneous systems to heterogeneous systems, for example communications and memory layouts. Additionally, the layout requires searching a large design space to achieve the most optimal system.

2.3 Hardware acceleration

2.3.1 The beginnings of hardware acceleration and the introduction of co-processors

Hardware acceleration is the process of implementing fixed hardware to perform a specific job. Hardware acceleration is a proven concept, and was being used as far back as 1979 [19]. Consequently it is a vast subject about which several books could be written. This section will focus mainly on themes that are addressed later in this Thesis.

Early hardware accelerators for Intel processors were the i80X86 family of devices which provided additional hardware to execute microcode on. The i80X86 devices had combined data and address busses. The i80X86 devices were intended for embedded systems, and so were formulated as microcontrollers rather than microprocessors. Later versions of the i80X86 family introduced more capability, particularly memory management applications. Additional hardware reduced the number of clock cycles per instruction, and increased the instruction set.

Intel also introduced a line of microprocessors designed solely for mathematical operations. An example is the i8087 device, which was also produced by IBM. It was capable of performing floating-point operations such as multiplication and division, but also more complicated operations like square-root, exponent and trigonometric functions. Performance improvements of up to 500% were reported using the i8087 over the i8086. Additionally, the power consumption of the device was remarkably low for its age. It consumed only 2.4 Watts while achieving 50,000 Floating-Point Operations Per Second (FLOPS). There have been many other mathematical co-processors since.

2.3.2 Modern acceleration

Hardware accelerators can be applied to many situations, from the very small, such as single mathematical functions, to the very large, such as data-centre acceleration.

Hardware acceleration is used to accelerate repetitive or intensive tasks or subroutines. A task, when broken down, will simply comprise mathematical functions. Most modern computers use floating-point numbers. It is well known that floating-point operations are intensive for GPPs, so mapping them to more suitable architectures vastly improves throughput and energy efficiency. This research presents two case studies for FPGA implementations - graphics rendering and neuron spiking models. A number of mathematical functions key to

these implementations have been identified, including: addition/subtraction, multiplication, division, square-root and exponential. Square-root and division operations are key for lighting calculations performed by graphics rendering pipelines, shown in Chapter 5. Non-linear functions, such as the exponential function, are fundamental to the transfer function of the Hodgkin-Huxley model of a neuron shown in Chapter 4. It is key to the research that these implementations are not taken from third party libraries so the exact characteristics of each implementation can be controlled for integration into the High-Level Synthesis tool presented in Chapter 7. Here the complex, and less well researched, mathematical function, square-root, shall be explored.

There are a number of iterative algorithms used to calculate the square-root of a number, such as Taylor-series [53], or Newton-Raphson [54, 55] approximation. The accuracy of an iterative method increases with every pass. For a processor this is a simple, although costly procedure, in terms of time and resources. Kwon *et. al.* compare a Taylor expansion based implementation of floating-point divide and square-root operations with Newton-Raphson and Goldschmidt methods in their 2007 paper [53]. The proposed architecture uses powering units, specialised hardware units that increase throughput and decrease latency of power operations.

Often the procedures must be re-worked to best suit the target. There has been research into a number of methods for square-rooting a number in hardware. Kachhwal & Rout presented a method using Vedic mathematics [56]. The method used a non-standard 24-bit floating-point format, requiring 90 slices, 99 slice flip-flops and 173 Look Up Tables (LUTs) on a Xilinx device. Xilinx FPGAs are broken into slices, each of these contains a number of resources - such as flip-flops, LUTs and carry logic. This is extremely low cost, however the implementation only achieved 1 – 1.25 MFLOPs for 91 mW power consumption.

Other algorithms to consider are multiplicative square-root [57, 58]. These take advantage of the DSP blocks on modern FPGAs to perform a number of multiply-accumulate stages. The requirements for DSP and memory cells make these methods not always suitable for intended applications.

Common square-root operations using hardware are based on the 'non-restorative' algorithm, equation (2.1). where D is the input number, Q is the quotient and R is the remainder. Chapter 4 describes the algorithm in detail.

$$D = Q^2 + R \tag{2.1}$$

The number of operations required is proportional to the width of the input number. Comparison operations are better for resource use and performance than multiply and divide operations required by iterative methods. The non-restorative approach is popular enough that there is a relatively large pool of research regarding the topic.

In 1996, an early implementation of the non-restoring method was shown in [59]. The algorithm was applied to 16-, 32-, and 64- bit unsigned, fixed-point numbers with pipelining added to increase throughput. It was compared to a Newton-Raphson based square-root algorithm. For the 16-bit input numbers the non-restoring algorithm took two fewer clock cycles. For 32-bit numbers it required one fewer clock cycles, however, for 64-bit inputs an additional eight clock cycles were needed. The benefits lay in the reduction in gates required. This demonstrates how intensive algorithms can be adapted to reduce cost.

Further research into the non-restorative method has been conducted to reduce latency as well as cost. Putra & Adiono presented a worst-case total path delay of 102.5 ns for a register-free, homogeneous square-root architecture using an Intel/Altera Cyclone II FPGA with 580 logic elements [60, 61]. The work presented used 32- and 64-bit fixed-point integers. The earlier work required registers but still had low resource cost and achieved a high frequency of operation: 110 MHz and 78 MHz respectively. The architecture did not use pipelining and had $N/2 + 1$ clock cycles of latency for an *N-bit* length input word. This reduced the overall throughput as it took more than one clock cycle to produce every answer. The 2014 work by Putra required one fewer clock cycles (latency of $N/2$) [60].

When using hardware implementations the power efficiency must be considered, as power availability is often limited. A low power, yet high speed implementation of a non-restorative square-root algorithm is shown in [62]. The research demonstrated total power for classical and reduced Newton-Raphson methods of around 200 μW. The proposed 8-bit non-restorative implementation used only 100 μW or lower. The proposed method also used less than 50% of the space of traditional methods.

Another common method for implementing the square-root function in hardware, or small microprocessors, is the COordinate Rotation DIgital Computer (CORDIC) method [63]. In the absence of hardware multipliers, or when gate count must be minimised, CORDIC methods yield high throughput, smaller implementations. Conversely, look-up table or power series based methods have a higher throughput than CORDIC methods when hardware multipliers are available. The CORDIC method is analogous to an analogue revolver that can operate in one of two modes: Rotation or Vectoring. Rotation computes the co-ordinates of a

given vector through an angle of rotation. In contrast, in Vectoring mode the magnitude and original vector are computed from the input co-ordinate components of the vector. There has been a large amount of research into CORDIC methods for computing other mathematical functions, for example logarithms and exponential functions, trigonometric functions, multiplication and division [64]. CORDIC based implementations for floating-point mathematical functions can be found in IP libraries from FPGA vendors, such as Intel FPGA.

It has been demonstrated that the square-root operation has been mapped to hardware by use of more hardware efficient methods. The implications of this will be considered further in Chapter 4. Other mathematical operations, such as the cube-root [65], can also be accelerated.

Hardware acceleration is not limited to individual functions. 'Big Data' can also benefit from hardware acceleration. Research was conducted by Microsoft into accelerating a server centre using a bed of FPGAs to create a reconfigurable fabric for 1,632 servers [66]. Adding FPGAs achieved a throughput increase of approximately 100%.

Mass file processing, such as in search algorithms, is time and resource intensive and requires careful memory management. Parallel processing can reduce the time taken for operations, especially using custom hardware architectures. Liu *et. al.* studied "pipeline based parallel frameworks for mass file processing" for cloud-based systems [67]. The work considered threads and homogeneous parallelisation concluding that "one pipeline made of three threads is more efficient than three homogeneous parallelisation threads" due to contention over shared resources. This demonstrated that, although dedicated hardware will speed up large data applications, the architecture must be carefully considered.

Data streaming [68] and analytics [69] are further examples of big data applications that can be accelerated using dedicated hardware. Big data tasks will probably involve sensitive data, so security is paramount. Shinde & Singh presented methods for extracting operations suitable for parallelism and ensuring security by considering data semantics and dependencies [68].

There is a large variety of applications that benefit from the inclusion of dedicated hardware. Once custom hardware methods have been proven using FPGAs, they can be used to emulate custom ASICs. The use of custom ASICs allows greater control over the process node technology, removes configuration overhead, tailors performance, and reduces power consumption, such as the threefold reduction shown between 65 nm and 40 nm process nodes [70]. However, creating custom ASICs is a costly process; therefore it may be more

beneficial to use off-the-shelf FPGAs. FPGAs are commonly implemented in the most advanced process nodes to offset the configuration overhead. Functions have been considered from individual floating-point maths operations to the handling of big data such as the internet. Other applications include data mining [71]; deep learning [72]; Phase Shift Keying (PSK) modems [73]; and video/audio synchronisation [74].

Some key themes have been observed, particularly an increase in throughput and a reduction in power consumption and resources. However, some limitations and design considerations are starting to emerge, especially into how data is managed and ensuring the security and synchronisation of data in parallel systems. The next Section will take an in-depth look into how hardware can be used to accelerate image and graphics processing techniques. This topic has been particularly highlighted due to the known computational intensity of the tasks. It also constitutes a main theme for some of the research carried out during the Ph.D.

2.3.3 Acceleration of image, video and graphics processing

As screens become higher resolution the need for processing larger quantities of data in the same time span increases. A 4K Ultra High Definition (UHD) image consists of 3840 pixels x 2160 lines. If each of these pixels is a High Dynamic Range (HDR) 10-bit colour, a single frame could contain 10 MB of data. Processing a single image is time-consuming enough; for video it may need to process 60 or 120 frames every second. Therefore image and graphics processing is a prime candidate for hardware acceleration.

Image processing applications consist largely of filtration, edge detection, pixel operations, geometric and orthographic transforms, compression, and colour space manipulation. The cost of these functions varies considerably. Often the higher intensity operations will require order statistics [75] or convolution kernels [76]. These are computationally expensive and are subject to change (filter weights, kernel size) across the image, which can lead to over-processing in direct hardware implementations.

In order to increase the efficiency of image filtering techniques, 'linear separability' is exploited. A filter is linearly separable when it can be split it into a series of simpler filters. These individual filters tend to be a lot more hardware efficient.

Another way to increase the suitability of image processing techniques is to exploit parallelisms. There are two levels of parallelism, image and operation that can reduce power consumption or resource use [77]. Jinghong *et. al.* used DSPs to integrate hardware into image processing applications [78]. DSPs offer an improvement in performance over GPPs, while

maintaining more flexibility than dedicated hardware. The flexibility of dedicated hardware can be increased using dynamic reconfiguration, which is discussed later in this Chapter.

Pham & Vliet exploited the concept of separability to perform bilateral filtering. This is a computationally expensive process as it must recalculate the kernel for every pixel [79]. Although the work presented did not have any implementation data, the premise presents a method for fast video preprocessing.

Preprocessing a video does alleviate processing demand on a system. However, often video must be processed in real-time. In order for this to happen at modern resolutions and colour depths, parallel processing paradigms must be exploited. Neoh & Hazanchuk began addressing these requirements in 2004. They presented a real-time edge detection system using a high powered FPGA device, achieving 4000 fps [80]. The hardware has since been surpassed.

Images and video data are computationally trivial compared to rendering graphics. In order to render graphics, a set of primitives must be transformed into visual representations tens or hundreds of times per second. Specialised GPUs have been designed with thousands of cores to handle the task. However, the architecture can be replicated on other, more flexible, devices. This allows task acceleration using specialised architecture.

A traditional graphics processor consists of an array of individual cores, memory interfaces and a threading engine, as was shown in Chapter 1. They achieve a rapid data throughput due to the Single Instruction Multiple Data (SIMD) architecture, allowing an array of data points to be processed simultaneously. Although a GPU is constructed to provide application-specific acceleration, it is still a collection of processing cores. As such introduction of dedicated hardware can still provide performance increase.

A large volume of current literature focusses on two directions: accelerating a single part of the graphic rendering process or replicating the SIMD processor using an FPGA.

Graphics pipelines can be thought of having three main sections: the vertex shader, the geometry processor, and the fragment shader. Shaders consist of mathematical functions, usually matrix based, to transform primitives or apply surface effects. The geometry processor culls covered objects, removes clipping and converts primitives to pixels.

Custom hardware, for example ASICs, can be extremely expensive. Hence using a generic platform, such as an FPGA, to provide acceleration for shaders seems reasonable. This was the premise of the work presented by Goddard & Stephenson [81] and Middendorf *et. al.* in [82]. These works demonstrate that, while it is true that FPGAs provide a versatile

platform suited to shader operations there are still limitations. Shader implementations attempt to mimic the construction of a GPU. This method was chosen as shaders are subject to change. However, abstracting away from the hardware causes performance to suffer.

Building the full graphics pipeline on a single FPGA device has been researched [83, 84, 85, 86]. Similar to some of the implementations of single shaders using FPGAs, these full pipelines produce co-processors from the FPGA's fabric. All of these works had similar end goals: a graphics processor built on a more flexible, low-cost device. Each implementation has its own specialities and limitations. The majority of the examples presented use OpenGL and Mesa libraries, making pre-existing code run on a number of different platforms. Processing speeds from 100 MPixels/second [83] to 278 MPixels/second [85] have been reported. Considering this research started in 2007, with the primary target being embedded systems, this performance is reasonable.

In 2014 Liu used OpenGL to produce a graphics renderer on an FPGA [84]. The system required an entire fixed-point mathematical library to be written. As FPGAs have very high fixed-point mathematical throughput, with consideration there is no loss in output quality caused by deviating from a floating-point system.

The works considered in this section have demonstrated the benefits of hardware acceleration. In order to mitigate the rigidity of hardware acceleration, research has studied using hardware-based topologies that more closely mimic GPPs, such as in the graphics rendering examples. Indeed, these solutions have proved successful and are adopted in industry. For instance D/AVE NX is a series of rendering core Intellectual Properties (IPs) for implementation on FPGA platforms that use the OpenGL Application Programming Interface (API) [87].

Addressing the limitations of hardware, particularly that of flexibility and size [81], is a key focus for the research presented in this Thesis.

2.4 Dynamic reconfiguration and context switching

The benefits of hardware acceleration have been demonstrated, but the drawback is the rigidity of such systems. The ability to change task is vital if hardware acceleration is to become mainstream. Tasks suitable for hardware acceleration are typically data processing oriented. This Section will expand upon the uses of dynamic reconfiguration, along with design considerations that must be made. This will be done in conjunction with context switching, task scheduling and acceleratable code selection.

Dynamic reconfiguration allows the configuration of FPGAs to be changed at runtime. Jahiruzzaman *et. al.* demonstrated how a reconfiguration method for FPGAs could be used to implement adaptable convolution kernels, instead of fixed implementations with parametrised components such as weights [76]. This requires each filter to be known and compiled at the design stage. The operation of both systems was the same. However, the selection of new parameters for a fixed implementation can be complicated. Yao *et. al.* presented methods of machine learning through use of an 'Evolution Algorithm'. An embedded host calculated new weights and passed them to the hardware [88]. Concerns were raised regarding efficiency of the evolution algorithm. It was suggested that future implementations may consider other bio-inspired algorithms such as particle swarm, ant colony and bee colony optimisation for greater efficiency.

Reconfiguring an FPGA at runtime has promise. However, it is time-consuming which potentially outweighs any gain obtained from using the hardware for acceleration.

2.4.1 Task scheduling

Generally the tasks being performed are a result of the user demands, so it is necessary to be able to schedule them at runtime as it is not know at the design stage. Lack of task scheduling reduces performance and can introduce bottlenecks.

Duplication can be an effective method to enable task scheduling on a multi-core processor [89]. The optimal replication of tasks is an NP-hard* problem. Using a processor to try and optimise its task scheduling, in addition to performing the tasks, adds additional overhead. Li *et. al.* presented an algorithm with order $O(v^3 log(v))$ (v is the number of nodes) to determine the optimal task scheduling approach, performing better than other algorithms [89].

While task duplication works on processors, it is less transferable to (heterogeneous) systems that use hardware acceleration. In this instance the requirement becomes isolating routines that would be best moved to the hardware. It has been shown subjecting tasks to hardware acceleration or replacing software routines with equivalent hardware yields significant performance increases, although it comes at a cost in terms of size and capital. Hence it is important to select the subroutines of an application that yield maximum return on investment. This requires additional information at the compilation stage.

A novel run-time approach for loop optimisation using Coarse Grain Reconfigurable Ar-

*Non-deterministic, polynomial-time hardness

rays (CGRA) is presented in [90]. A 'Greedy' approach was implemented to produce optimised binaries describing processors that can be loaded and executed at runtime. It is claimed that the optimisation is better than an off-line compiler for a 16-issue Very Long Instruction Word (VLIW) processor when performing scheduling. The use of a VLIW processor abstracts away from the hardware, reducing performance. However, the methodology is promising for on-the-fly compilation, although it is not yet mapped to fine grain reconfigurable architectures, like FPGAs. It is still the case that an off-line approach for selecting optimal hardware accelerators for FPGAs may have to be used.

Prakash *et. al.* [91] have researched automatically selecting appropriate subroutines to be translated into hardware. The work focussed on creating custom instruction sets, particularly in area-constrained FPGAs. Previous work by Prakash *et. al.* detailed an "FPGA-aware custom instruction enumeration and selection technique". An expansion was designed to maximise the logic utilisation in a confined FPGA design space. An area-time metric was derived, which allowed the selection process to avoid compromising on the final solution's quality. This created the best cost/performance trade-off. It was noted that when "available FPGA area is limited, it is prudent to choose smaller and frequently occurring patterns that can efficiently utilize FPGA space as they provide more performance gain per unit area". Therefore, sometimes optimising smaller tasks, such as floating-point maths functions, would be a better use of the FPGA.

Variables are often required in multiple locations; should the application be hardware accelerated, two processes could both require access to the same variable. Prakash *et. al.* studied the use of local memories in accelerated systems [92]. Having separate caches for the processor and the hardware requires expensive cache-coherence and Direct Memory Access (DMA) operations. The method proposed by Prakash *et. al.* removed the need for DMA transfers, producing an architecture where a single local memory serves both the processor and the hardware accelerator. This is shown in Figure 2.6. While both hardware and software requiring access to the memory, the hardware component makes many more access than the software. The research presented in [92] demonstrates a 47% increase in speed for a Secure Hash Algorithm (SHA) implemented this way. The method used custom instructions and memory operations.

Prakash *et. al.* published a third paper that modelled the communications overhead for accessing memories from hardware accelerators [93]. Unlike the previous work, which implemented un-cached memory structures, this modelled the read/write access penalties of

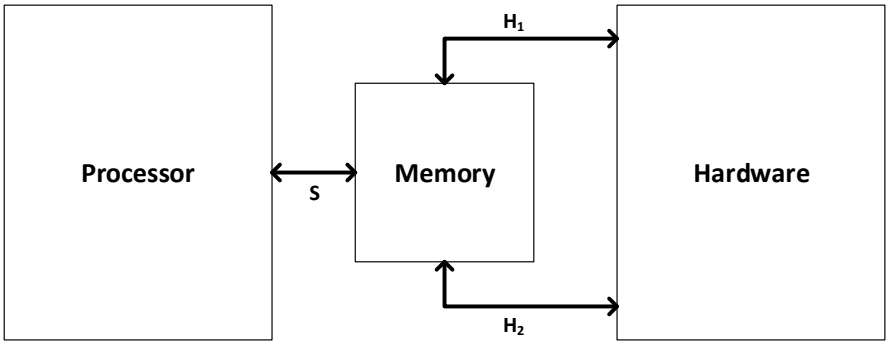


Figure 2.6: A local memory is accessible by both the hardware accelerator and the rest of the application. For optimal performance the number of accesses made by the hardware to memory must be far greater than the number made by the software, $S << H_1 + H_2$, in cases such as bitcoin mining.

processor local memories. An accurate performance model could then be made. It would be suitable for automating design space exploration for heterogeneous systems with hardware accelerators, thus determining optimal system design.

2.4.2 Dynamic reconfiguration

Dynamic reconfiguration allows different hardware accelerators on the same FPGA, dependent on the current environment. There are a number of implications when designing for a partially reconfigurable system. At the beginning of this research, these systems were still in their early conception. Design flow and techniques were still in early development.

There are two types of dynamic reconfiguration: full and partial. Full reconfiguration blanks the entire FPGA then loads a new, complete configuration. Partial reconfiguration is more complex, requiring a combination of static and reconfigurable logic. The static logic is never reconfigured and must be integrated with the reconfigurable regions. This presents problems at the boundaries between the two. It is necessary to manage communications between a region that is operating and a region that is being reconfigured. Other concerns are how to separate a design into reconfigurable and static regions and how to manage and execute the reconfiguration process.

Di *et. al.* studied three existing dynamic reconfiguration technologies and performed a case study using an encryption system [94]. Dynamic reconfiguration resulted in an improvement of 40% for resource optimisation. The work used partial reconfiguration tools available from Xilinx. The results showed how to implement a bus that communicates between static and reconfigurable logic. The paper lacked a concrete design flow for all tool suites. However, Di *et. al.* proposed a set of steps for bus macros that span reconfigurable regions using the

Xilinx tool suite.

There are additional challenges for dynamic reconfiguration. Diessel noted that challenges surround the validation, testing, and run-time system behaviour verification of dynamically reconfigurable systems [95]. Diessel conducted work into increasing the model accuracy for state-of-the-art reconfigurable systems to address these problems.

Dynamically reconfigurable systems are subject to change, increasing the difficulty of thermal floor-planning. Pagano *et. al.* developed a thermal-aware floor-planning tool that considered descriptions of the heterogeneous resources and partially reconfigurable constraints [96]. The tool models thermal effects on a system with simulated annealing and mixed integer linear programming. The results showed a reduction in power consumption and peak temperature in several designs.

To make full use of dynamic reconfiguration, new configurations should be synthesised at runtime. This introduces considerable overheads associated with place and route procedures. A method that circumvents the place and route overhead is proposed by Diessel & Maskell with the use of pre-defined blocks [97]. The proposed pre-routed cores have several features for run-time fitting operations. They have a consistent interface for co-location that allows them to be connected instantly, reducing fitting times. Additionally, they are scalable to allow them to use the maximum available bandwidth of a system. This approach increased the flexibility of generic reconfigurable FPGA systems. Chapter 6 will discuss a method for runtime synthesis, optimisation and compilation of a Higher Level Language (HLL).

2.4.3 Context switching

Task scheduling and selecting the optimum parts of processes to undergo hardware acceleration allows the development of high performing heterogeneous architectures. Modern processors perform a number of tasks seemingly at once. This is done using context switching to allow the processor to move the contents of the stack to memory so another process can be run. Later the old process is returned to as though there has not been any interruption. Running processes heterogeneously limits the ability to context switch. To overcome this, a method for context switching hardware must be researched.

Context switching and pre-emption have been widely researched, albeit for different uses. Pre-emptive techniques for processors are well founded, allowing time-multiplexing of CPUs. It is becoming common to move from ‘soft’ context switching solutions that use a processor and OS, to using ‘hard’ systems with dedicated logic. Dedicated logic systems introduce over-

head from additional hardware but have significant benefits. Rafla & Gauba demonstrated saving and reloading 12 registers of a MIPS processor [98]. When using internal registers to perform switching 49% more data was processed.

Sawalha *et. al.* added 'context sets' to a multi-core processor to achieve low latency switching [99]. The main drawback was that the number of context sets exceeded the number of cores, therefore the resource cost grew with the number of device cores. The additional cost was justified since only a couple of clock cycles were required to switch the executing threads for all the cores, compared to thousands of cycles required using the OS. However, the hardware added system constraints. There was an increase in latency to flush out all processor pipe-lines. Additionally, thread assignment was dominated by the number of context sets integrated into the processor.

The use of processor caches conflicts with context switching, causing context switch misses. Context switch cache-misses occur due to the perturbation effect from a program being restored to find some of its working set in the cache has been overwritten. Re-order misses tend to peak when cache perturbation affects approximately half the cache [100]. Hardware systems can forgo the use of a caches in favour of data pipelines, therefore not becoming subject to data perturbation. The development of hardware context switching systems is discussed further in Chapter 6.

Context switching is not limited to GPPs. Other architectures, such as GPUs, present further challenges for context switching. There is a high dependency on the kernel program to control the context switching. This is not necessarily deterministic nor will it be able to support all pre-emption techniques. Lin *et. al.* demonstrated a lightweight approach to context switching in [101] that uses compiler and hardware co-design to enable instruction-level pre-emption on Single Instruction, Multiple Thread (SIMT) architectures. Using the proposed method of compression and analysis, register size and latency were reduced by almost 92% and 60% respectively, compared to more complex approaches. The work by Lin *et. al.* demonstrated that if a system has a kernel there is considerable overhead for context switching. On average the context switch takes place in 4.0 μs (2,800 clock cycles). Therefore, context switching, even on advanced architectures, is still expensive and time-consuming.

There are potential benefits from implementing context switching in hardware accelerators. The flexibility of heterogeneous architectures can be increased by context switching accelerators associated with current processor tasks. However, only certain tasks can be

context switched. Primarily they must be **mutually exclusive** of each other, for instance cryptography, software defined radio and protocol processing applications that are not attempting to run in parallel with dependencies on each other.

2.4.4 Applications for dynamic reconfiguration and context switching on FPGAs

Dynamic reconfiguration and context switching increase the flexibility and performance of FPGA-based systems. These systems have many applications. Adding reconfiguration to the embedded hardware further increases the desirability and number of potential applications, although it introduces overheads.

Data mining is currently a topic of interest, especially since Bitcoin and Ethereum were introduced. It is computationally expensive for a processor to perform. Dynamic reconfiguration can allow spare FPGA resources to be used for performing these types of background tasks.

Using FPGAs to accelerate data mining applications in embedded hardware is presented by Perera & Li [102]. An FPGA was used for matrix computation. Reconfiguration allowed three different implementations on one device, at different times. Without having all three implementations running concurrently, the total power efficiency was shown to increase.

Langenbach *et. al.* used FPGA resources provided by 'System-on-Chip' (SoC) devices for task acceleration. The research evaluated the performance of the Linux Kernel FPGA Framework [103] and found that it provided low runtime overhead (<2%) when using an FPGA as a Linux kernel accelerator. However, concern was raised regarding the initialisation and deinitialisation time of the FPGA.

Reconfiguration can also be used in less obvious applications. Cetin *et. al.* explored the use of dynamic reconfiguration to recover from Single Event Upsets (SEUs) by reconfiguring corrupt FPGA portions [104]. Experimental results showed that modular reconfiguration is more responsive and less energy consuming than scrubbing techniques. However, the overall configuration required higher area, ran more slowly and had increased design complexity.

The control of event-driven robots using context switching techniques was presented in [105]. Furthermore, a method for context switched security (vNative) was presented in [106]. vNative was as a way to freeze unauthorised applications in a 'Bring Your Own Device' scenario.

Life and safety-critical scenarios, such as military applications, are areas where context

switching can further aid [107]. In these applications it is critical that the appropriate information is delivered quickly to the operator and is able to mitigate human error. A level of 'system self-learning' can also be employed to help refine systems for greater performance. Reinforcement learning systems, such as image and handwriting recognition, can use context switching to load the appropriate context in quickly changing environments [108].

Digital Audio Broadcasting (DAB) receivers are a conventional application for dynamic FPGAs. Sections of the receiver are partially reconfigured to produced resource-efficient implementations [109]. The work presented by Feilen *et. al.* discussed the use of context-aware data framing to reduce context switching overhead. The work they presented does not consider performing context switching in the style of a processor.

The literature presents a prominent role for FPGA acceleration in heterogeneous computing. There are some design limitations and complexities associated with this technology that need to be overcome. System design, verification and thermal planning are straightforward challenges. More complex problems include communications and integration with GPPs. The developments that have already been made to GPPs that allow task scheduling and multiple thread handling should be kept. However, they may reduce gain from dedicated hardware accelerators. Context switching needs to be performed with low latency to prevent bottlenecks. Identifying which application routines are to be accelerated is important, especially as there is currently a limitation on run-time synthesis and device fitting.

2.5 Code compilation and automatic optimisation techniques

Writing code that performs optimally on any architecture is almost impossible. Modern devices are so complicated that compilers and tool-chains are used to apply optimisations.

A key optimisation for software is to handle loops in the most parallelised way possible. HLLs support loop operations. Unrolling loops can result in hundreds or even thousands of operations. A single-cored processor would have to execute these in sequence. However, there are a number of techniques for reducing the impact of loops on throughput. Additionally, using multiple cores can further exploit parallelisms to optimise data flow.

There are a number of optimisations that can be used to achieve dynamic data flow management on modern processor topologies. Santiago *et. al.* examined three methods: *stack tagged data flow*, *tag resetting*, and *loop skipping*. The benefits and drawbacks of each were reported [110]. Each method optimised the number of loop iterations by enabling simultaneous execution of instructions with different loop iterations. Each operand was tagged as it

became available, and the associated loop was executed as soon as all operands were present. The three optimisations all had different impacts on the stack. *Stack tagged data flow* reduced control overhead by using stacks of tags. For nested loops the overhead was increased. *Tag resetting* allowed a return to zero whenever it was safe, reducing cost. *Loop skipping* was used on loops of pre-determined length to avoid stack comparison. This final technique is more interesting when considering heterogeneous architectures with hardware acceleration. Loops of determinable size are prime candidates to be moved from the processor and onto the hardware. The paper concluded that a hybrid approach that used all of the optimisation techniques produced the best results.

There are constant developments being made to optimise compilers, especially for parallel computing. Techniques such as simulated annealing are giving way more to bio-inspired and artificial intelligence approaches, like the random walk method shown in [111]. Loke & Wang showed that using 'random walk' allowed on-line iterative generation of permutation matrices that helped the compiler to converge to an optimised answer faster.

These optimisation techniques are for traditional processor topologies, although some techniques can be applied to heterogeneous architectures. Optimisations are also not limited to loop unrolling and iterative generation on processors. It was reported that specialised architectures, such as GPUs, show improvement from optimised code generation [112]. Optimising for the architecture reduced code size and execution cycles by 10.3% and 16.8% respectively. Similar techniques can be used for optimising hardware synthesis from HLLs.

2.5.1 Coarse grain reconfigurable architectures

Reconfigurable targets are either Coarse Grain Reconfigurable Arrays (CGRA) or Fine Grain. CGRA consist of a large number of functional units that are connected together to perform a task. Fine grain arrays consist of individual logic blocks and look up tables. The different devices have different uses and therefore different programming methodologies. However, they are both parallel devices. This is their most appealing attribute.

Common methods for programming CGRAs use graph descriptions. DFGenTool, presented by Mukherjee *et. al.* [113], generated a data flow graph from a HLL input that can be synthesised and implemented on a CGRA. The DFGenTool uses the DOT (a graph description language) format, a common CGRA programming method that is human readable. Additional design constraints can be implemented after the automatic generation.

Creating a completely optimised output from a HLL for a reconfigurable architecture

is challenging, especially for an unconstrained application. The DFGenTool is designed to optimise for loops in the input sequential language and to map them to the CGRA architecture in a parallel implementation. This cannot necessarily also optimise for scheduling, which is where the additional input of the designer comes in once the DOT output has been generated.

Mi *et. al.* produced an Automatic Parallel Module (APM) [114], designed to generate sub-functions and then schedule their execution to further exploit the architecture. The APM was demonstrated to provide better performance than single-threaded execution, achieving an almost two times speedup in some cases.

CGRA provide a platform suited to mapping HLLs to. The construction of a CGRA with higher level functional blocks over a fine grain architecture removes some of the complexity when applying optimisations. Implementing applications with a high level of parallelisation on a CGRA leads to a performance increase.

2.5.2 Hardware compilers

Fine grain architectures offer more control over implementations. The additional control adds additional design complexity, particularly for High Level Synthesis (HLS) tools. Programming languages for FPGAs (Hardware Description Languages (HDL)) differ to conventional software languages. HDLs are used to describe the desired architecture for the algorithm being implemented. However, software languages describe the order in which instructions from a pre-existing instruction set should be executed to achieve the desired outcome. There are a number of tools that generate HDL from HLLs.

The intended uses, benefits, and limitations of a number of these tools were analysed by Yankova *et. al.* [115]. The tools have since been improved, but it still provides a good starting point. Yankova *et. al.* analysed three tools: *SPARK*, *ROCCC*, and *DWARV*. *SPARK* and *ROCCC* had been in development for several years more than *DWARV*. *SPARK* provided users with a toolbox of functions designed to optimise area and delay with a particular focus on operation scheduling. This exploited instruction-level parallelisms for control-intensive kernels. *ROCCC* was aimed at streaming applications, with its main focus being on optimising loop parallelisms and memory access. Since *ROCCC* was designed for streaming application it worked on the premise that data can be windowed - operate on snapshots of the incoming data. *DWARV* offered no optimisations; it was purely targeted as a straight conversion from HLL to Very High Speed Integrated Circuit Hardware Description Language (VHDL), with considerations for software/hardware co-operative architectures. *DWARV's*

lack of out-of-the-box optimisations made it a more appealing choice for integration into other HLL synthesis projects where developers could apply their own optimisations.

The largest criticism of *SPARK* and *ROCCC* was that for optimisations to be applied the designer needed to input additional parameters. This meant designers needed to understand hardware as well as software. A primary role of these tools is to abstract away from the nuances of hardware to make it simpler for software developers to design for them without additional training.

With the increased move towards heterogeneous architectures and the co-execution of programmes on both processors and hardware, toolchains such as *DWARV* that synthesise for this environment become more useful. Bertels *et. al.* proposed an early evaluation *hArtes* toolchain which focussed on co-design and co-verification for hardware/software environments [116].

It has also been shown in this Chapter that hardware accelerators can perform a key role in alleviating computational load from recursive functions. Middendorf *et. al.* presented work for the automatic generation of recursive functions in hardware through a technique termed 'partial stream rewriting' [117]. They propose an algorithm that identifies and handles recursion and indirect calls using a single type of rule. The simplicity of the rule allows hardware with support for dynamic thread creation, parallel recursion and data dependent branching to be synthesised.

2.5.3 Example applications for generated HDL

There are a number of existing applications that used HLS. The automatic synthesis of microcode for VLIW processors [118] is a relatively mature and well-established task. Most ASICs or Application Specific Integrated Processors (ASIPs) are formed from some sort of VLIW processor. Designing these devices from scratch every time would be time-consuming. Instead, similarities in them can be exploited. Kobayashi *et. al.* presented a synthesis tool that allowed certain parameters, such as the number of slots, pipeline stages and instruction behaviour, to be specified with a HLL [118].

Lattuada *et. al.* presented an aerospace system that was generated using HLS [119]. The researched used an already available synthesis project, *TASTE* [120], and extending the functionality with *Bambu* [121]. Add-ons produced a tool capable of generating hardware implementations from languages such as C, and automatically integrating them into the aerospace system. Given the complexity and safety regulations of the aerospace industry this

is a substantial achievement.

The automatic synthesis of combinational circuits has been considered. Previous discussions noted that FPGAs can be susceptible to SEUs, particularly in high radiation environments like space. Partial reconfiguration has been shown as a way to repair the region of the FPGA configuration that has developed an error. This requires the configuration files for sections of the target device. Gorodilov proposed a method for automatically generating these 'combinational circuits sets' using a tool that can take a single VHDL implementation and generate a set of failure modes [122]. From these, new partial reconfiguration files can be automatically generated.

2.6 Summary

This Chapter has explored literature regarding several key topics: processor technology development, system architectures, hardware acceleration, task scheduling (including dynamic reconfiguration and selection of application software to accelerate) and compiler optimisations (particularly focussing on the automatic conversion of HLL to hardware). The discussions have not been exhaustive due to the diversity of topics and the sheer volume of available literature.

Processor topologies have undergone large development since their early inception as 4-bit microprocessing devices. A need for task-specific devices was identified and addressed, for example GPUs and DSPs. There is diminishing return with every new development - seen in the decline of Moore's Law - leading to new types of processors being researched, such as photonic processors. These are unlikely to be a reality for some years to come.

Architectures can be classified as homogeneous or heterogeneous, each of which has different advantages and limitations. Spreading computational load over a number of different cores has performance benefits, but communication overheads and memory accesses introduce bottlenecks [123].

It has been shown that hardware acceleration has performance and efficiency benefits. Hardware accelerators can cover a range of applications. Integrating these with GPPs creates heterogeneous system architectures. Prakash *et. al.* researched selecting routines suitable for being accelerated to give the greatest performance increase. The results showed that accelerating frequent and small operations can bring about the biggest overall system gain, particularly in area constrained situations. Additionally, new memory topologies for custom architectures were used to mitigate the communication overhead.

Dynamic reconfiguration allows FPGA configurations to be changed at runtime. Implementing hardware accelerators on FPGAs that can adapt to the current environment increases the overall system performance and flexibility.

Context switching techniques are implemented in traditional processors to allow tasks to be swapped without loss of data. Similar techniques will be translated to hardware accelerators in Chapter 6.

Finally, the complexity of producing the hardware designs has been introduced. Designing for hardware, especially in hardware/software co-operation systems, requires specialisms in both fields. Automating the process helps to reduce time to market and increase performance and reliability. HLS tools attempt to optimise the conversion process, but are still limited and require input from the designer.

From the literature it can be concluded that heterogeneous architectures that use flexible hardware for accelerators could increase system performance. The next Chapters will present work on the development and application of accelerators, introduce dynamic reconfiguration techniques, apply techniques commonly found in processors to hardware and present the design for a HLS tool that uses a OpenGL Shading Language (GLSL) input.

Chapter 3

Hardware Implementations of Fundamental Maths Functions

Chapter 2 presented information on hardware acceleration and the implications this has for hardware-based co-processing. Hardware accelerators ranging from the very small to the very large have been discussed.

The question now becomes how to make the most efficient use of hardware accelerators. The interaction between hardware and software requires careful planning. The situation becomes more complicated when memory accesses and different clock domains are introduced.

This Chapter will implement a number of fundamental mathematical operations as hardware floating-point accelerators. A number of floating-point mathematical functions are required to implement an OpenGL based graphics processor and a neuron model on an FPGA. These functions will be implemented to create a library for use in this research. They will be tailored for the presented case studies in Chapters 4 and 5, the context switching architectures presented in Chapter 6, and for use with the high-level synthesis tool presented in Chapter 7. For silicon implementations, floating-point number formats do not necessarily provide the most efficient implementation, [124]. Instead Constantinides asserts that custom data formats provide a better solution. Despite their inefficiency, there are a number of open-source and commercialised projects that provide floating-point IP cores for FPGAs. They can be found in Xilinx [125] and Intel FPGA [126] libraries, among others. The libraries may include FloPoCo (formally FPLibrary) [127], VFLOAT [128], and CORDIC based methods. Implementations of floating-point functions from these libraries can support varying precision and are optimised for certain parameters - area, throughput, power. As a result their flexibility is limited. These libraries are not used in this research to allow complete control over

the implementation of the floating-point function. It also allows for better integration into the applications and tools presented in future Chapters. The use of standard floating-point formats allows easy communication between flexible FPGA technology and fixed processor architectures.

Floating-Point Units (FPU) have been included in processors for many years, allowing difficult mathematical operations to be off-loaded to more suitable hardware. The hardware was designed to perform particular floating-point operations more efficiently than the host processor could manage, hence improving the overall system performance. However, any dedicated hardware accelerator has its limitations, the most obvious being on flexibility.

In this research the hardware is replaced by reconfigurable logic, FPGA fabric, in a heterogeneous arrangement with an embedded processor. Using the FPGA fabric to implement the floating-point accelerators allows for the replacement of FPUs within processor implementations. The silicon area for the processor can be reduced without compromising on floating-point performance.

3.1 The implementation of algorithms

Programs consist of data flow and control flow. An algorithm describes the order in which data is manipulated. Moving from a conceptual understanding of a program to its implementation usually requires several stages. Modern day programmers can choose from a number of different programming languages. However, all languages are reduced to the same thing in the end: the instruction set of the target architecture.

Consider a basic function: calculating the result of two numbers undergoing basic arithmetic. The user enters numbers and the operation to be applied, which are stored in memory. This requires some memory operations to load and store the information, and a mathematical operation to process the numbers and create the output. Using this as the basic premise, more complicated programmes can be realised by combining memory operations, Boolean and mathematical functions.

High Level Languages (HLLs) allow difficult or repetitive tasks to be expressed in easy-to-read instructions. These are compiled down to the available instructions for a given processor. Modern processors are complicated, but, their instruction set is essentially just memory and mathematical instructions.

In a simple Instruction Set Architecture (ISA) the number of instructions is very limited. Memory operations could include move, load, and store, and arithmetic operations

could include add, subtract, and divide. The most basic systems are Minimal Instruction Set Computers (MISCs), which contain only fundamental instructions from which all other instructions are built. If a processor architecture does not have a floating-point maths block, the equivalent instruction requires a cascade of operations to achieve the result.

Early methods for enhancing the capability of a processor provided a separate coprocessor designed to perform floating-point operations. In the 1980s Intel launched the i8087 floating-point coprocessor to work alongside the i8086 microprocessor, which could achieve a performance increase in excess of 500%. Processors include floating-point blocks. The additional hardware requires additional instructions which increases the flexibility and performance of the device. However, this does not always mean the architecture is the most optimal for the program being implemented.

3.2 Processors versus dedicated hardware

Processors operate on binary numbers that can represent any type of data, such as numbers, characters, or strings. Regardless of the type of the data, the stored information is always binary. It is the interpretation of a number that makes it a floating-point number, for example (shown in Figure 3.1). This particular interpretation breaks up the binary data into sections. The floating-point interpretation increases the representable numeric range. The real value of the floating-point number is calculated using equation (3.1). The Bias of a floating-point number offsets the exponent, allowing both positive and negative exponents. Representing floating-point numbers in this way provides a sliding window of precision to increase the representable range.

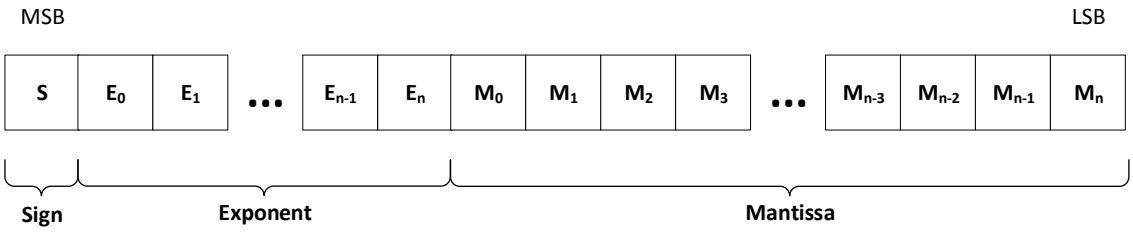


Figure 3.1: Floating-point numbers take a standard width word (e.g. 16-, 32- or 64-bit) and split it into a sign, exponent and mantissa. This is used to represent a number with increased range over fixed-point binary. MSB - Most Significant Bit, LSB - Least Significant Bit.

$$(-1)^{sign} \times (1.mantissa) \times (Exponent - Bias) \tag{3.1}$$

For a processor to perform the operations to a floating-point number it must process the binary word to extract the different floating-point elements, then perform the operation and reconstruct the floating-point number. Combinations of integer operations can be used to emulate floating-point operations in software. This is a complex and intensive process.

In certain areas the question as to whether floating-point emulation is more or less practical than including a hardware-based floating-point unit is still under debate. Additionally the efficiency and performance of fixed versus floating-point operations are being examined. Assertions have been made regarding how to ensure the highest performance from a processor. In 1996 Kraeling stated that using a fixed-point system "is the single largest improvement a programmer can make in C to reduce execution time" [129]. In the general case, for an architecture with no FPUs, this holds true. The amount of processing required for the floating-point numbers compared to the fixed point versions is large. However, having a fixed-point-only system has disadvantages, such as the range and precision of stored numbers.

Applications ranging from signal processing to graphics rendering will require the floating-point system. Even considering just the basic quadratic formula, equation (3.2),

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{3.2}$$

it can be seen that without the extended precision of a double floating-point system, depending on the problem compound error quickly makes the result unusable. Power, square-root and division operations can result in large or small results that cannot be stored in lower precision formats. There is the demand for high-performance computing, so a compromise must be found.

Research into methods for converting floating-point functions and designs into a fixed-point equivalent exists, [130, 131]. As designs become larger the time taken to pass information between devices starts to impact significantly on runtime. Although difficult to implement, there is the immediate advantage of being able to perform all calculations quickly using the same device, such as removing the need to communicate between two different devices.

There are limitations to floating-point emulation. The proposed method for implementing floating-point algorithms on fixed-point, Single-Instruction Multiple-Data (SIMD) architectures presented in [131] relies on custom instruction and data formats. This increases design complexity and time to market, and reduces portability between devices. Traditional hardware-based floating-point coprocessors still present the best option for processing floating-point numbers. They are well understood and introduce little additional overhead.

Floating-point operations can be parallelised using dedicated hardware. This allows the processing of the exponents and mantissa to occur simultaneously, minimising the number of required clock cycles. Dedicated hardware can also provide native support for bitwise operations, making processing certain elements of a word much faster.

Hardware accelerators are still relatively expensive to implement, and can only provide a certain level of acceleration. The acceleration of a large number of tasks currently requires large co-processors, for example the Intel Xeon Phi modules.

Having the ability to implement acceleration in a more flexible medium is of great interest. FPGAs are renowned for their ability to implement a design with very high performance and efficiency. This typically results in a co-processor implemented on FPGAs, such as the vector/scalar design presented in [132]. These designs tend to trade-off ease of implementation with throughput when compared to dedicated data processors.

Despite the complexity of implementation, the FPGA is a promising platform for accelerating mathematical operations. The FPGA reduces the hardware costs therefore removing the need to implement operations in software.

This Chapter will examine the implications of implementing floating-point maths operations directly on FPGAs. The implementations will start with basic maths functions (add, subtract and multiply) from which more complicated functions can be realised. Subsequently, more complicated functions such as square-roots and exponents will be considered in Chapter 4.

3.2.1 How can hardware make life better?

In Chapter 2 a number of architectures were presented. The concept of the 'best' architecture was also considered. It was concluded that the 'best' architecture is dependent on the situation in which it is used. For instance, a mobile platform may not require high-performance computing, instead having strict power limitations. In this case a low power architecture is the 'best'. Similar arguments can be made for area, power, and efficiency.

Implementing floating-point operations using hardware increases the performance due to the inherent native support for bitwise and parallelised operations. Furthermore, it offers the designer more control over aspects such as the area, performance, and efficiency of a system.

3.3 Implementing floating-point mathematical operations on a hardware architecture

Mathematical functions range from the conceptually simple, such as *add*, to the complicated such as *exponential*. Comprehending the simple functions is straight-forward, making their implementation easy. However, complicated functions, like the exponential, become more convoluted and their implementation becomes more complicated. In order to realise these functions, they must be broken down into a series of simple operations. Complicated approximations such as *Newton-Raphson*, *Taylor-series expansion* and *Euler* work by breaking down the operation into simple functions.

The IEEE-754R Standard [133] provides a comprehensive list of all functions that should be implemented in floating-point format. The standard covers the form the operations should take and how to indicate exceptions. The functions implemented for this research are not the complete list of functions detailed in IEEE-754R; rather, they have been selected as being important for the FPGA implementations provided in Chapters 4 and 5.

3.3.1 Basic mathematical functions in hardware

The functions that have been included for analysis as 'basic' functions are: add/subtract, multiply, greater than, less than, floating-point to fixed-point, and fixed-point to floating-point.

Functions have been implemented using a Hardware Description Language (HDL). HDLs offer a number of pre-existing functions, much like any procedural language. In addition, they also natively support ease of working with individual bits or groups of bits from registers. Register widths can be selected to match the needs of the module, rather than being fixed widths as they are in processors. This flexibility allows resource and performance optimised designs to be created.

Floating-point modules make use of integer operations where possible due to the FPGA's high integer performance.

The following Sections will describe the operation of the modules. All modules are pipelined to increase the performance and provide an issue rate of one (a new answer is provided on every clock cycle). This can be done for relatively little resource cost.

3.3.2 Analysis methods

All hardware designs were subject to automation, allowing significant numbers of implementations to be tested under a wide range of scenarios. Important design metrics from compilation and simulation were extracted. The metrics were then processed to produce graphs and tables ready for comparison and analysis.

Determining the functionality of a floating-point operation was achieved by comparing results to an IEEE-754R compliant processor (Core-i7 7700HQ) and measuring the error. The error is given as a comparison between the results of the hardware implementation and the processor. The error is measured in two ways: normalised error (ϵ) and relative error. Normalised error is given by equation (3.3).

$$\epsilon = \frac{S_p - S_h}{S_p} \tag{3.3}$$

Where S_p is the answer calculated by the processor and S_h is the answer calculated by the hardware implementation.

Relative error is given in Units of Least Precision (ULPs). A single ULP represents the smallest expressible value of a floating-point number for a fixed exponent value. Relative error in ULPs was calculated by comparing the two floating-point results. The exponents were equalised before calculating the difference in the mantissas, as shown in Figure 3.2.

3.3.2.1 Add/subtract

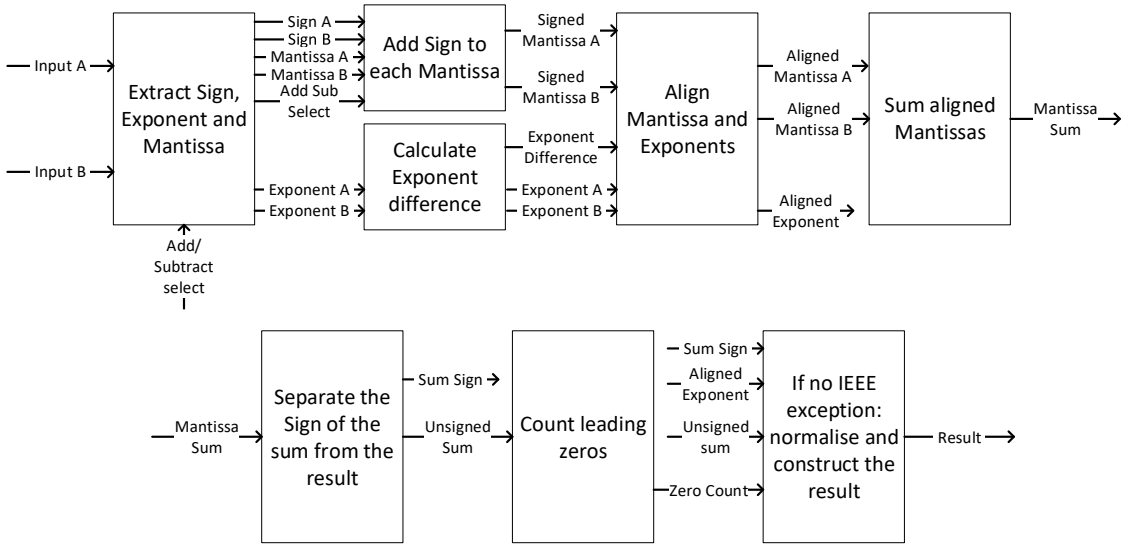


Figure 3.3: Flow diagram of the stages performed by the hardware implementation of a floating-point add/subtract module.

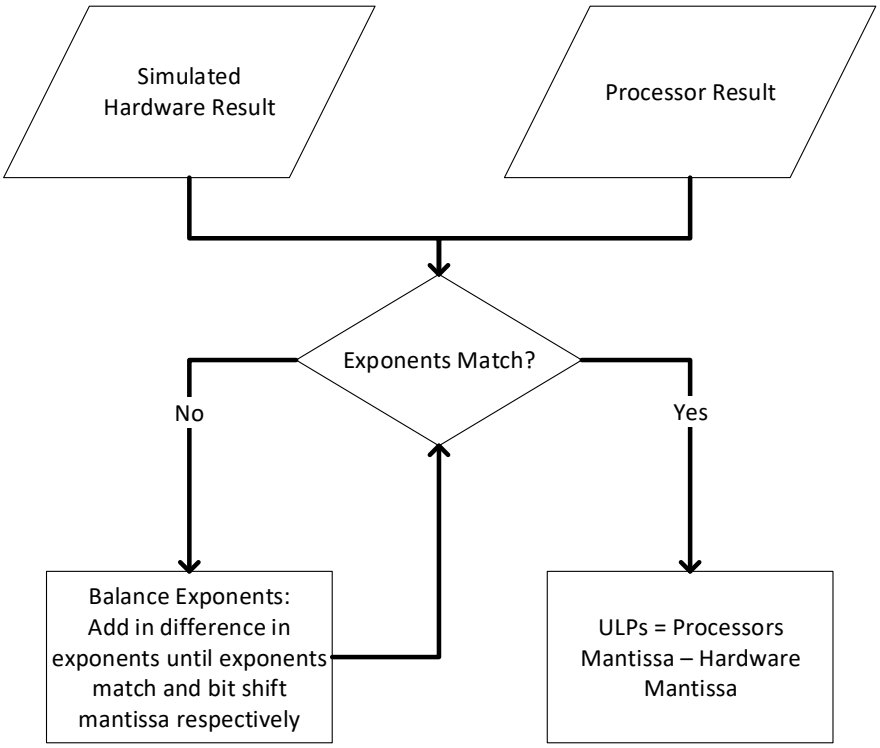


Figure 3.2: Relative error requires the exponents of both floating point numbers to match. If this is true the number of ULPs is given by a difference in the mantissas. If the exponents do not match, one of the floating point numbers may be changed by simple addition until they match so long as the mantissa is bit-shifted to compensate. Increasing the value of the exponent requires a bit shift right; decreasing the exponent requires a bit shift left.

Figure 3.3 shows the flow of the addition/subtraction routine, which are accomplished using almost identical logic. A select line changes which operation is performed. The input numbers are broken into their component parts: *sign*, *mantissa*, and *exponent*. The difference in the exponents is calculated. If the exponents are not the same magnitude, the mantissas are aligned by bit-shifting by the difference in the exponents. Each mantissa is converted to a signed integer. The greater of the two exponents is selected as it represents the exponent for both mantissas.

The aligned mantissas are summed and the result is broken back into its sign and an unsigned result. The number of zeros in the unsigned result is counted to determine the location of the first '1'. This information is used to normalise the mantissa of the floating-point number by bit-shifting the result left. The sign of the output comes from the sign of the summed mantissas. The exponent of the output is the larger of the input exponents.

3.3.2.2 Multiply

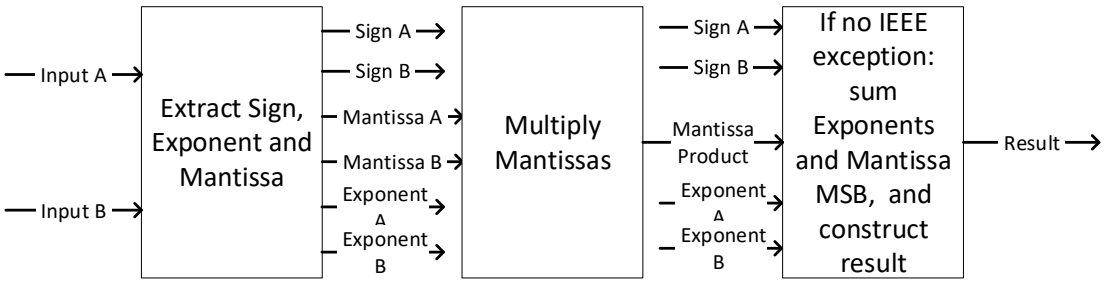


Figure 3.4: Flow diagram of the stages performed by the hardware implementation of a floating-point multiply module.

Figure 3.4 shows the flow of the multiply routine. Similar to addition, the floating-point multiply starts by de-constructing the input numbers into their component parts. The input mantissas are multiplied together, and the result is used to construct the output. The exponent of the output is the sum of the input exponents and the MSB of the result of the mantissa multiplication. The sign of the result is the XOR of the input signs. Finally, the output mantissa is selected by checking the highest bit from the mantissa multiply operation register. If this is true, the system must include the MSB of the mantissa multiply operation in the result and loses the LSB. Otherwise the MSB is lost and the LSB is included. Unlike addition, a floating-point multiply operation is very quick to perform.

3.3.2.3 Compare

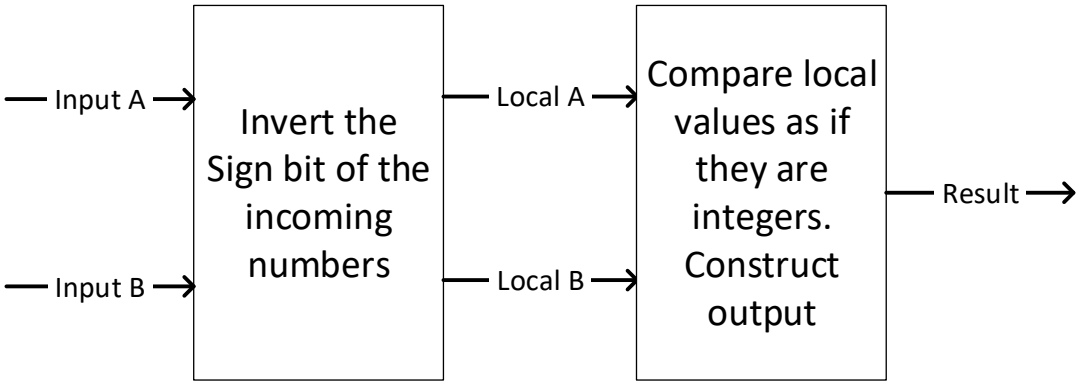


Figure 3.5: Flow diagram of the stages performed by the hardware implementation of a floating-point compare module.

Figure 3.5 shows the flow of the compare routine. Due to the arrangement of a floating-point number, a compare operation is easily achieved. HDLs allow operations on individual bits of

registers, which is used to invert the sign bit of the incoming values. The effect of inverting the sign bit is to create two registers that can be compared as though they are fixed-point integer values - the exponent and mantissa, Figure 3.1, no longer need to be evaluated separately, as in equation (3.1). The output must have the sign bit inverted again to comply with the floating-point standard.

3.3.2.4 Typecast

In HDLs there are no 'typed' variables, only registers. Only the interpretation of the variable by the designer gives the data its type. Due to some of the applications that are demonstrated in later Chapters, it is important to be able to convert between data 'types'. Two 'typecast' modules that convert the contents of registers between the fixed-point and floating-point data representations were developed.

3.3.2.5 Floating-point to fixed-point

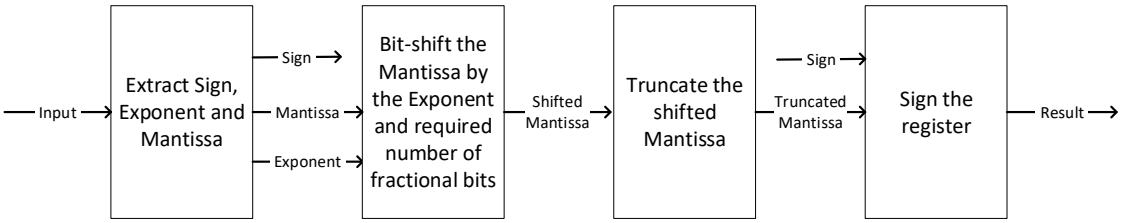


Figure 3.6: Flow diagram of the stages performed by the hardware implementation of a floating-point to fixed-point module.

Figure 3.6 shows the flow of the floating-point to fixed-point routine. This is the easier conversion. The input is split into its component parts. The mantissa is bit shifted up or down by an amount represented by the exponent. The result could then be converted to two's complement signed by examination of the input sign bit.

3.3.2.6 Fixed-point to floating-point

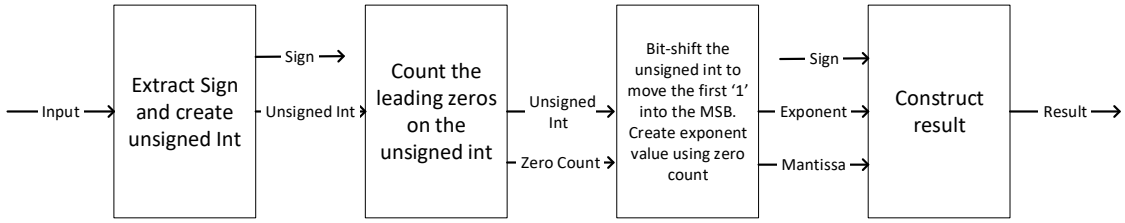


Figure 3.7: Flow diagram of the stages performed by the hardware implementation of a fixed-point to floating-point module.

Figure 3.7 shows the flow of the fixed-point to floating-point routine. To convert from fixed-point to floating-point the sign of the input number has to be removed. The remaining portion is converted to an unsigned register. If the sign was negative a bitwise inversion is needed; otherwise there is no change. The number of leading zeros for the unsigned register is counted. This information is used to bit shift the unsigned register to create the mantissa, and is the exponent for the floating-point representation.

3.3.2.7 Compliance with IEEE-754R

Most commercial processors comply with IEEE-754R [133]. IEEE-754 details the way in which numbers are stored; how the binary word is broken down to represent different parts of the floating-point number; and the special cases resulting in certain operations. This was later updated to IEEE-754R with the introduction of the half-precision or 'float16' format, shown in Table 3.1.

As the modules presented here are to be used as hardware accelerators for processor routines, compliance with the IEEE-754R standard is vital. This ensures that if an operation results in a special case, for instance *Not-a-Number* (NaN) or $\pm$*infinity* (Inf), it can be handled correctly before being passed back to the processor.

Table 3.1: Bit configurations for IEEE-754R floating-point number formats.

Format	Sign	Exponent	Mantissa	Bias
Half-precision	1	5	10	15
Single-precision	1	8	23	127
Double-precision	1	11	52	1023

3.3.2.8 Analysis of resource cost and error

Note: In this research tables show the resource count as reported by the Quartus fitting tool Version 15.0. Numbers outside of the parentheses refer to the total number of elements used by the hierarchy, while numbers in the parentheses refer to the amount of that particular resource used at that level of the hierarchy.

There are a few metrics that are important to consider: resources, throughput, and error. Depending on the target architecture for which a design is synthesised, the resource count and maximum operating frequency can vary. By synthesising the designs for a single device, meaningful comparisons between different mathematical function implementations can be made. Table 3.2 presents resource use and maximum operating frequencies when synthesised for an Intel Cyclone V 5CSXFC6D6F31C6N. This is a low-cost FPGA-SoC device containing a dual-core ARM Cortex-A9 processor. Additional information for resource use and operating frequency of the designs synthesised for single- and half-precision floating-point numbers can be found in Appendix A.

Intel FPGAs consists of Adaptive Logic Modules (ALMs). Each of these ALMs contains an eight-input fracturable Look Up Table (LUT), two adders and two registers [134]. The LUT can be split into a number of different arrangements to best suit the operations being performed and use as few resources as possible. Typically LUTs are used with four- or five-inputs, however the eight-input fracturable LUT can be configured with three-, six-, and seven-inputs as well. Intel eight-input combinatorial LUTs can be divided between two Adaptive Look-Up Table (ALUT).

The tables that report resource use for the Intel devices show a number of different values of ALM used. The total ALM use is a combination of ALMs used by the final placement operation, ALMs that can be recovered using dense packing operations and ALMs that remain unavailable - for instance from routing restrictions or the fitter being unable to pair half-ALMs together due to the number of inputs used. Hence it is the 'ALMs Needed' column that is of most interest.

The basic operations presented here can be broken into mathematical functions and non-mathematical functions. Add/subtract and multiply will be considered as mathematical functions. Greater than, less than, floating-point to fixed-point, and fixed-point to floating-point are comparative or typecast operations. There is a notable difference in the number of resources required by the mathematical functions over other functions.

Ideally the resource use should be kept as minimal as possible. The smaller the number of

Table 3.2: Resource requirements and timing analysis for simple maths functions that do not require other functions to implement. Implementations are using double-precision floating-point accuracy.

Module	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs Recoverable by Dense Packing	[C] Estimate of ALMs Unavailable	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
Add/Subtract	1034.5 (1034.5)	1487.5 (1487.5)	465.0 (465.0)	12.0 (12.0)	1099 (1099)	2280 (2280)	0	106.87	103.99
Multiply	241.0 (241.0)	328.0 (328.0)	87.0 (87.0)	0.0 (0.0)	408 (408)	521 (521)	4	107.03	101.84
Multiply with no DSP	1414.5 (159.2)	1305.0 (236.6)	90.0 (88.2)	199.5 (10.8)	2418 (179)	521 (521)	0	82.56	81.85
Greater than	124.0 (124.0)	134.5 (134.5)	10.5 (10.5)	0.0 (0.0)	191 (191)	194 (194)	0	188.39	192.09
Less than	123.5 (123.5)	132.5 (132.5)	9.0 (9.0)	0.0 (0.0)	191 (191)	194 (194)	0	172.98	174.58
Float to integer	237.0 (237.0)	256.0 (256.0)	20.0 (20.0)	1.0 (1.0)	340 (340)	261 (261)	0	162.34	161.92
Integer to float	332.5 (332.5)	374.0 (374.0)	42.5 (42.5)	1.0 (1.0)	491 (491)	332 (332)	0	109.02	107.2

resources used per module (atom), the greater the number of hardware-accelerated functions can be implemented on the same device. Additionally, smaller accelerators can have a greater f_{max}. The addition function, Section 3.3.2.1 has the highest number of stages and contains performance limiting stages such as integer addition and leading zero counting making it have a higher resource use. The compare operations (Section 3.3.2.3) and typecast operations (Sections 3.3.2.5 and 3.3.2.6), have very few stages. They can be accomplished using primarily bit-shift and bit-select operations, both of which are very resource light, leading to small atoms.

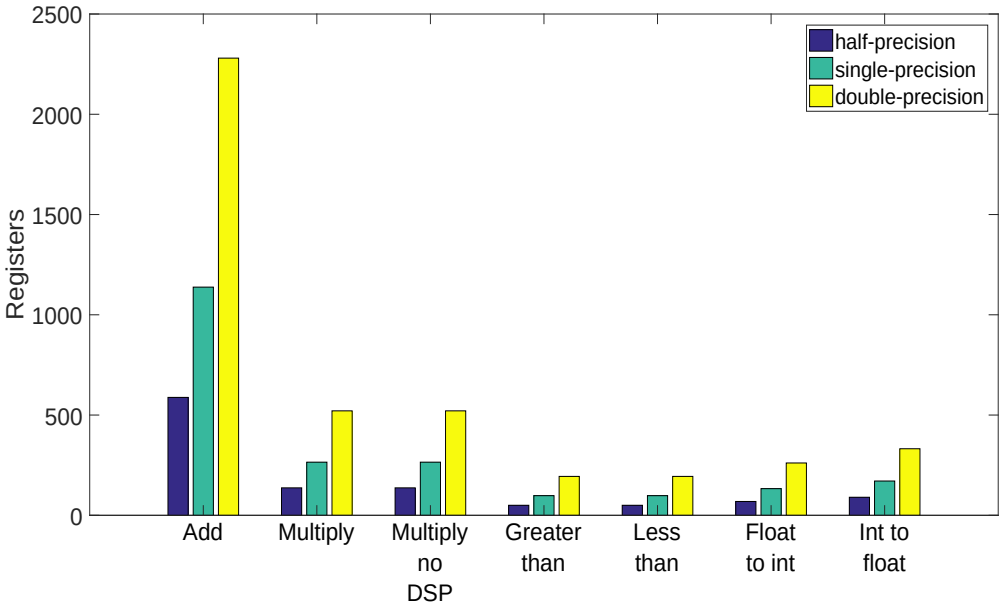


Figure 3.8: Registers required for hardware implementations of fundamental mathematical operations.

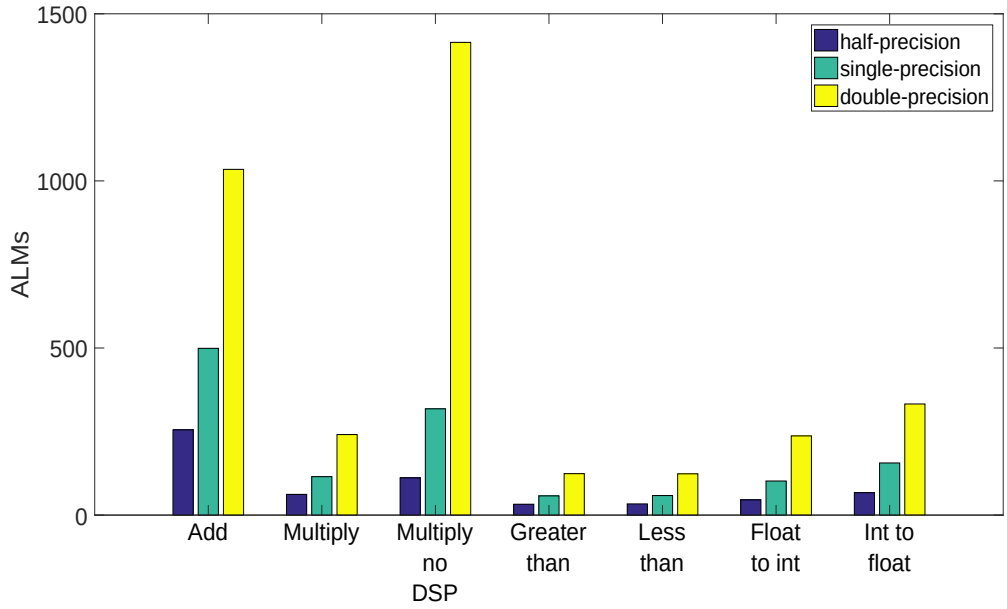


Figure 3.9: ALMs required for hardware implementations of fundamental mathematical operations.

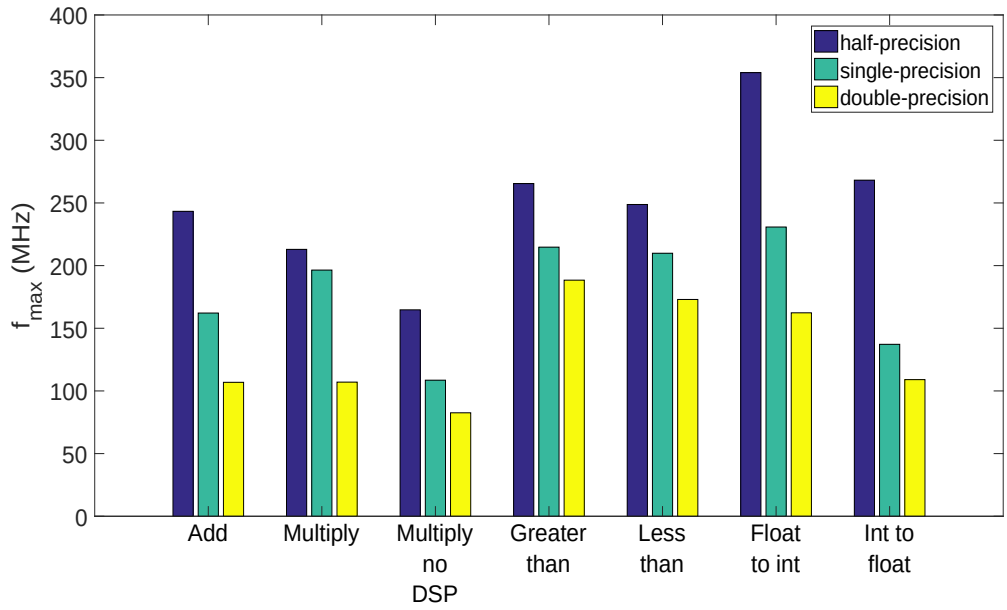


Figure 3.10: f_{max} of hardware implementations of fundamental mathematical operations.

From Table 3.2 it can be seen that Digital Signal Processing (DSP) blocks have been used for the floating-point multiply. A double-precision implementation used four DSP blocks, while single- and half- used only one (Appendix A). The use of DSPs is not problematic: the inclusion of DSP blocks in FPGAs nominally allows the acceleration of specific functions and uses fewer resources. In this case the DSP blocks are used for an integer multiplication of the two input mantissas. DSP packing (allowing the synthesis tool to use the DSP blocks on the

device) can be turned off. This allows analysis of the effects of including the DSPs. Table 3.2 shows that the use of DSP blocks allowed the synthesis engine to save around 1200 ALMs and 2000 ALUTs. Although this will vary slightly between target devices, it demonstrates the ability of dedicated architectures to save floor space and resources. The majority of modern FPGAs from all vendors will include hard Intellectual Property (IP) blocks such as DSPs for this purpose.

The resource use figures given in Table 3.2 demonstrates how intensive double-precision floating-point operations are to perform in hardware. The introduction of the half-precision floating-point format greatly reduces the number of required resources to implement functions (Appendix A). The trade-off is the reduction in precision and range that can be represented, although for the majority of applications this does not pose a problem.

Figures 3.8 and 3.9 show a side by side comparison for the different fundamental maths functions at the three different floating-point precisions. These graphs show how costly floating-point addition is compared to floating-point multiplication. Allowing the use of DSP blocks reduces the number of ALMs for a multiply operation to approximately 40% of an addition operation. However, it is also important to consider the overall system requirements for the final implementation. Allowing use of DSPs for multiplication greatly reduces the number of ALMs needed for this function but DSP blocks are a valuable resource that may be better used in other parts of the system.

Intel provides a library of floating-point functions for implementation on their FPGAs, such as the Cyclone V used in this Chapter. The library includes an implementation of an add/subtract function and a multiply function that can both be implemented in single- or double-precision. The functions provided by Intel FPGA have a non-zero latency value, meaning they have an issue rate grater than one, whereas the implementations in this Chapter are all designed to have an issue rate of one. The add/subtraction function has a latency of twelve clock cycles in single-precision and 20 clock cycles in double-precision. The Intel FPGA multiply function has a latency of six and eleven clock cycles for the respective precisions. The resource count for the Intel FPGA add/subtract module is 880 LUTs in single-precision and 2235 LUTs in double-precision. A single ALM for an Intel device contains an eight-input fracturable LUT. The resource count for the implementation of the add/subtract module given in this Chapter is 499.0 ALMs and 1034.5 ALMs for the respective precisions. Similarly, the Intel floating-point multiply core in single-precision mode requires 286 LUTs and two hardware multipliers (DSP blocks); in double-precision the requirements are 848 LUTs and 8

hardware multipliers.

The implementations in this Chapter for a floating-point multiplier are given with and without DSP blocks. For implementations allowing DSP blocks to be used, a single-precision multiplier requires 115 ALMs (1 DSP block) and a double-precision multiplier requires 241.0 ALMs (four DSP blocks). Disabling the DSP blocks significantly increases the ALMs required to implement the module, 318 ALMs and 1414.5 ALMs respectively.

The other key metric presented here is the maximum operating frequency of the modules. A combination of f_{max}, Figure 3.10, and issue rate (number of clock cycles required between valid results) determines the throughput of the implementations. Throughput is expressed in Floating-Point Operations per Second (FLOPS). Since each implementation does not re-use resources, the throughput is **exactly the same** as the maximum operating frequency. The addition and multiplication modules achieve just over 100 MFLOPS in a double-precision implementation. **Reducing** the precision of the implementation, **increases** the throughput. When a half-precision adder is implemented the throughput becomes 246 MFLOPS and a multiplier becomes 205 MFLOPS. The Intel FPGA floating-point implementations allow the user to select the target frequency of operation. Increasing the target frequency consumes more logic and increases the latency. When the core is set to a higher f_{max}, the latency increases. If the design has a dependency on latency, the throughput will decrease. However, applications will attempt to de-couple the dependency between latency and throughput, so as to reduce the impact on maximum throughput. The throughput, therefore, becomes defined as the maximum operating frequency divided by the issue rate (number of clock cycles between valid data) of the module.

In summary, the benefits of compromising on range and precision can be seen in both resource use and performance.

The implementations for floating-point functions detailed in this Chapter perform reasonably, but they could be optimised significantly more. Indeed, adding just two pipeline stages to the double-precision floating-point multiplier yields an increase in operating frequency of more than 20%. FPGAs are highly parallelisable, deep pipeline devices; since the floating-point operations are acyclic the pipelining can be extended to almost any depth. This optimisation can be achieved at very low area cost since FPGAs typically have flip-flops associated with the Look-up Tables (LUTs); these flip-flops would otherwise go unused.

This Chapter discussed a number of open source projects that provide floating-point libraries for FPGA implementation. The functions in these libraries are the result of consid-

erable research into techniques for minimising resource use and increasing throughput.

There are a number of optimisations that may be implemented in libraries. Using a leading zero detection algorithm for mantissa alignment in the floating-point addition/subtraction module presented in this Chapter has very high resource use (over 2200 registers and 1000 ALMs for double-precision). The latency of a floating-point adder can be greatly reduced by using barrel shifters, and incorporating techniques to combine rounding and significand addition operations [135]. Other techniques for improving floating-point adder design include Leading One Prediction (LOP); far and close data-path algorithms [136]; and end-around carry techniques [137]. LOP predicts the position of the leading one in parallel with the 2's complement adder, rather than detect the leading one after the add stage. The LOP method reduces the overall latency but at the cost of area. Far and close data-path algorithms are used to determine when a leading zero count is necessary, significantly decreasing latency.

Similarly, there has been research into increasing the performance of other floating-point operations such as splitting expensive multiply operations into a number of smaller parts, and exploiting partial products [138]. It would be possible to apply similar optimisations to the floating-point implementations presented by this Chapter. However this research is more focused on how the presented implementations are used; particularly in conjunction with the synthesis tool presented in Chapter 7.

3.3.2.9 Analysis of error

It has been shown that implementing these basic mathematical functions on an FPGA leads to relatively small sets of logic, with relatively high throughput. The size can be further reduced by reducing precision and numeric range. To validate the functionality of each module the error is measured.

According to IEEE-754R [133], a successful floating-point operation is defined as having **one ULP or less** relative error. Plots of relative and percentage error for the double-precision adder and multiplier are given in Figures 3.11 and 3.12 respectively. (See Appendix B for graphs of single- and half-precision error.)

Both the addition and multiplication functions have an error of one ULP or less over a valid range. This makes them acceptable as implementations of floating-point functions as per IEEE-754.

The multiplier, exhibits a large step change in error beyond a certain point. This is an artefact of the number system. The range and precision of a floating-point number are given

by the expressible range of the exponent and length of the mantissa. For a given combination of inputs, the output of the multiplier can quickly exceed the representable range of the number system. For inputs that result in an invalid number in the output, the error becomes large. However, as this is an artefact of the number system, these results are removed. The *x-axis* on Figure 3.12 has been reduced to represent an input range where the output results in a valid entry for the number system. Numbers that are not representable by the number system are given as 'Inf' or 'NaN'.

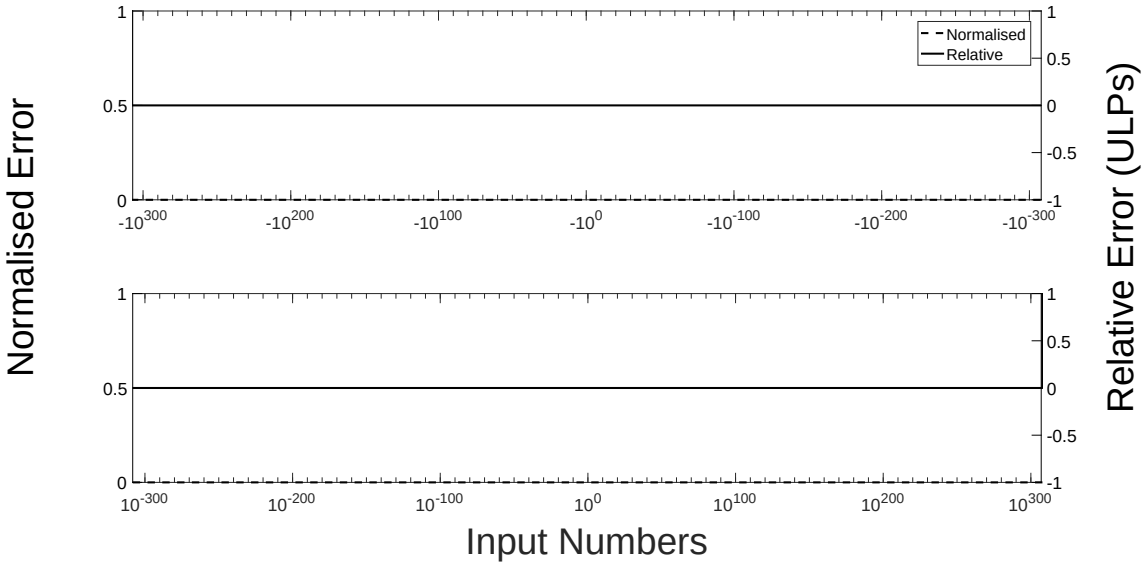


Figure 3.11: Relative and absolute error for hardware adder in double-precision. The top graph is negative input number. The bottom graph is positive input numbers.

The construction of floating-point numbers is given in equation (3.1) in Section 3.2. The value of the highest bit of the mantissa is given as 2^k, where k is the Exponent minus the Bias. Therefore the lowest value the mantissa can represent is 2^{k-L_m} where L_M is the length of the mantissa in bits. This is referred to as 1 ULP. IEEE-754R defines the maximum allowable error that can be inserted by a floating-point function as 1 ULP. So long as the output of the function satisfies equation (3.4) (that is the output of the function is representable by the number system) the total error from the output of a floating-point mathematical function can be calculated using equation (3.5). If the equality in equation (3.4) is not met, the error is given as $\epsilon_T = \infty$.

$$\text{Output}_{\text{min}} \geq f(A, B) \leq \text{ Output}_{\text{Max}} \tag{3.4}$$

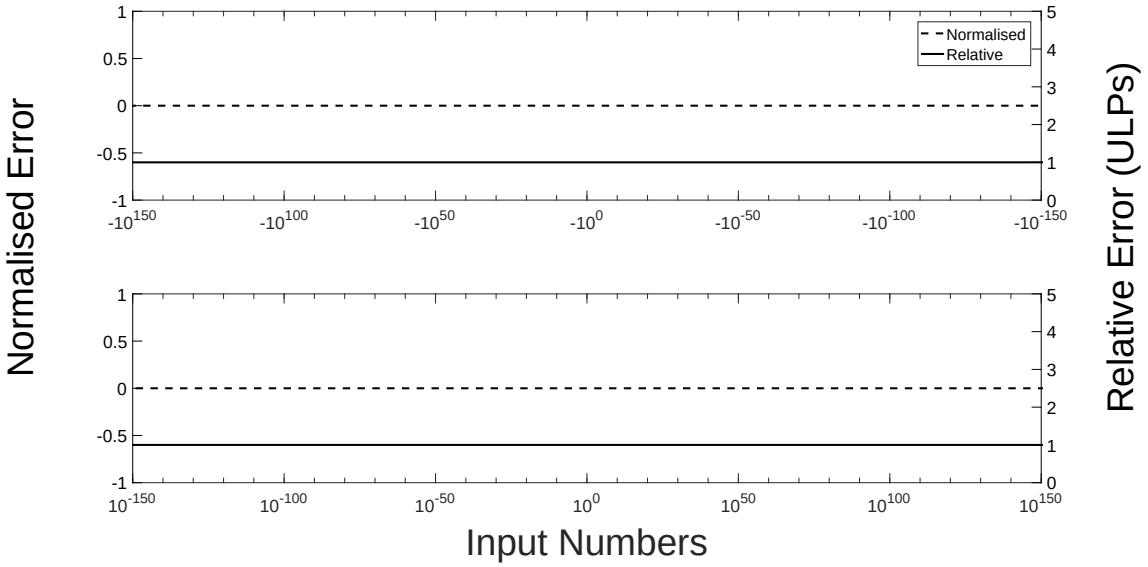


Figure 3.12: Relative and absolute error for hardware multiplier in double-precision. The top graph is negative input number. The bottom graph is positive input numbers.

$$\epsilon_T = \epsilon_A + \epsilon_B + \epsilon_f \tag{3.5}$$

Where $\epsilon_{A,B}$ is the relative error of the input values, ϵ_f is the relative error added by the mathematical function, defined as being one or fewer ULPs (2^{k-L_m}), and ϵ_T is the total error in the output value of the function.

The implementations of floating-point functions from this Chapter satisfy these error conditions.

3.3.3 Vector and matrix operators

While mathematical functions that operate on one or two scalar values are important, they are only a small subsection of the functions that are possible. This Section will consider vector and matrix operations.

Vector and matrix operations are a collection of addition, subtraction, multiplication, and division operations performed a number of times. The output matrix is formed by iterating through each element of the input matrix. This allows easy construction of the vector/matrix functions using the functions discussed previously.

Despite this relative ease of design, several important factors - resource use and throughput - must be considered. Their implications shall be made clear in Chapter 5. A design for a matrix/vector multiplier will now be considered. Other matrix/vector functions can be

constructed in a similar way.

The module can be optimised for either resource use or performance, or a combination of the two. Optimising the design for performance requires each operation to have a dedicated block of logic. Consequently the result is large, but capable of delivering a new result vector or matrix on each clock cycle. Alternatively, to conserve resources, fewer floating-point adders and multipliers are used. Feedback allows resource re-use. This will reduce the size, but also the throughput of the implementation.

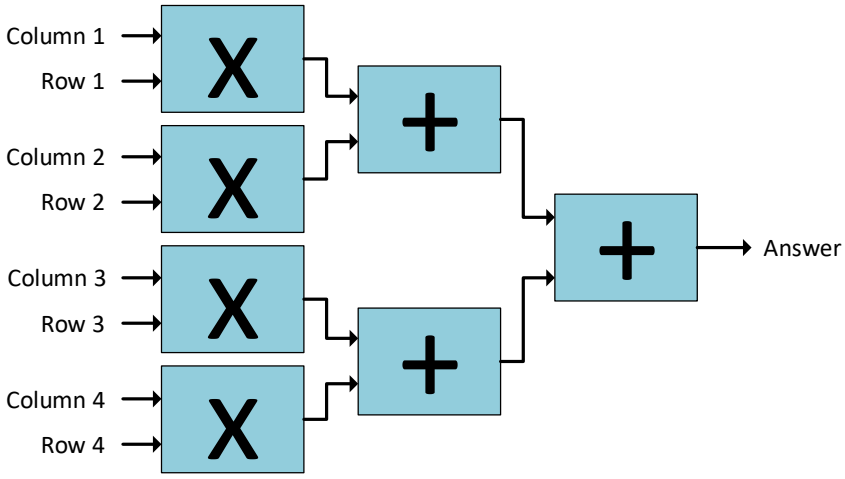
Using the matrix/vector multiplier function as an example, Figure 3.13 shows methods for both resource and performance optimisation. For any given $m \times n$ matrix and $n \times 1$ vector, there are mn multiplications and $m(n-1)$ additions required to calculate the result. Performance optimisation, shown in Figure 3.13a, generates each of these blocks separately. For large values of m and n, this approach uses large numbers of resources. Alternatively, resource optimisation can require as little as one multiplier and one adder. Figure 3.13b shows a resource optimised solution can perform the same operation. This method requires the incoming data to be registered locally and then iterated over until the entire result has been calculated. If the input matrix width (n) is greater than two, a feedback path around the adder is needed to sum the total number of elements in each row after the multiplication operation. This can be scheduled to reduce execution time by using some registers and a multiplexer. The latency of the adder and multiplier is fixed, therefore, it is possible to schedule operations to minimise the number of unused clock cycles into the adder block.

Which form of implementation to use depends on the system. Using a more performance optimised implementation offers better throughput but at a cost. Chapter 5 demonstrates how performance optimisation is not always necessary and that other parts of the system will govern the best type of implementation to use.

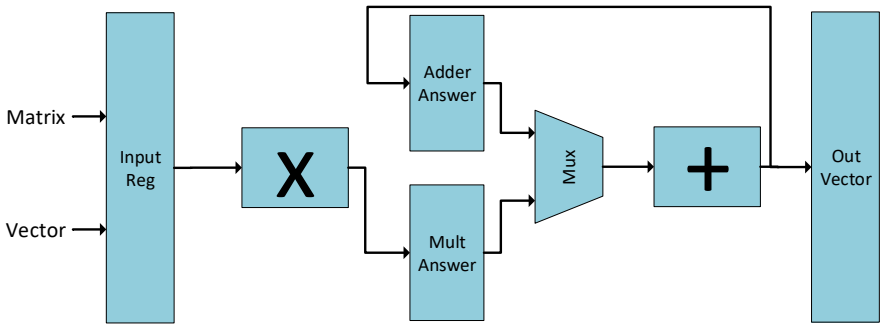
Table 3.3: Resource requirements and timing analysis for floating-point matrix and vector operations commonly performed by a GPU. Implementations are using double-precision floating-point accuracy.

Module	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs Recoverable by Dense Packing	[C] Estimate of ALMs Unavailable	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
Normalise	21852.4 (23.5)	35572.5 (23.5)	14215.4 (0.0)	495.4 (0.0)	13506 (42)	70982 (34)	24	97.73	95.49
Dot product	1648.4 (102.4)	2233.8 (115.5)	609.0 (22.8)	23.5 (9.7)	2059 (117)	4007 (164)	12	98.69	96.29
Vector Length	18748.4 (291.3)	31787.1 (316.9)	13332.7 (39.5)	294.0 (13.9)	10820 (376)	62309 (166)	12	100.17	97.02

There are a number of matrix or vector operations that can be accomplished using simple



(a) Performance optimised



(b) Resource optimised

Figure 3.13: Calculating the result of a matrix/vector or matrix/matrix multiply requires a series of multiply and add operations. To optimise this for performance the multiply and add blocks can be replicated a sufficient number of times, resulting in an issue rate of one, 3.13a. This can produce very large hardware designs and it may not be necessary to run the function at such a high rate. In this situation it would be more prudent to reuse the resources and iterate through the data, Figure 3.13b.

addition or multiplication operations. Additionally, there are also more complex functions, such as those involving square-roots. Table 3.3 shows resource cost for three functions: normalisation, dot product and vector length. These are matrix/vector operations that are commonly used in graphics rendering. Implementations shown here use double-precision floating-point format. (Appendix A shows the resource use for single- and half-precision floating-point formats.)

Matrix/vector operations, often require potentially complicated functions. The more complex mathematical functions are covered in Chapter 4. Decisions about resource re-use versus performance optimisation become more important. The 'best' implementation depends on the system and application. The location of the operation relative to the system bottleneck is important. Deriving the most efficient system implementation is complicated;

methods to automate this are presented in Chapter 7.

The matrix/vector operations (normalisation, dot product and vector length) will now be discussed in more detail. The vector length function needs a square-root function. This is very resource intensive. Normalisation uses the vector length function, with a division stage at the output. Division operations are also complex, have a high resource and time cost and are expensive to perform, requiring iteration. Both the square-root and division operations are covered in more detail in Chapter 4. The dot product function is achieved using only multiplication and addition stages. This leads to a much smaller implementation for this operation compared to normalisation and vector length, Figures 3.14 and 3.15.

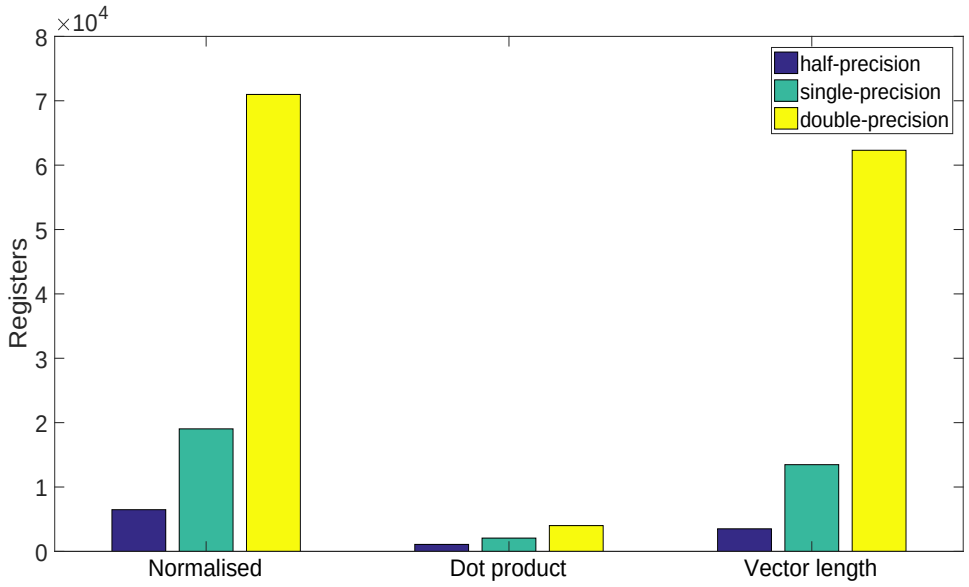


Figure 3.14: Number of registers required for hardware implementations of some example vector operations.

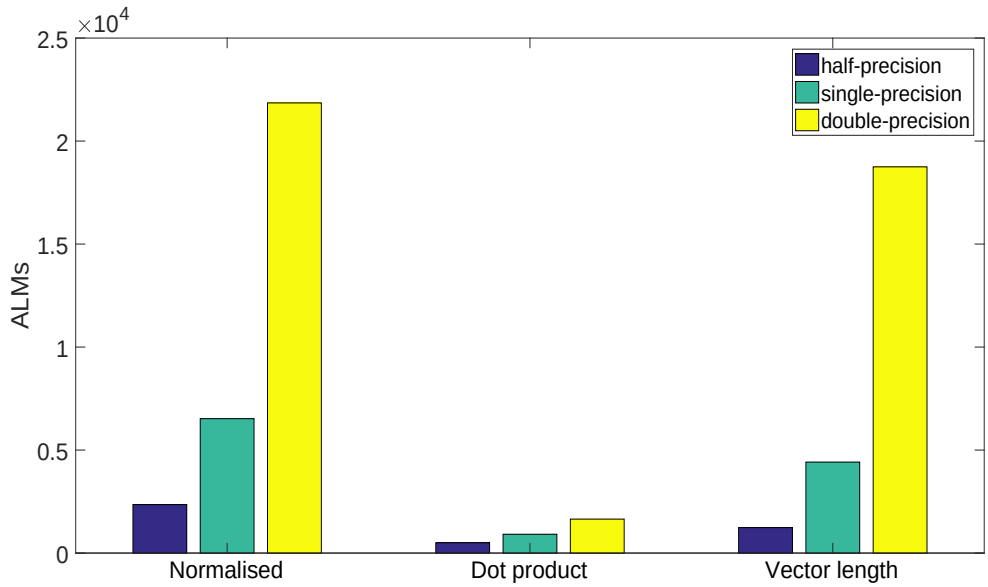


Figure 3.15: Number of ALMs required for hardware implementations of some example vector operations.

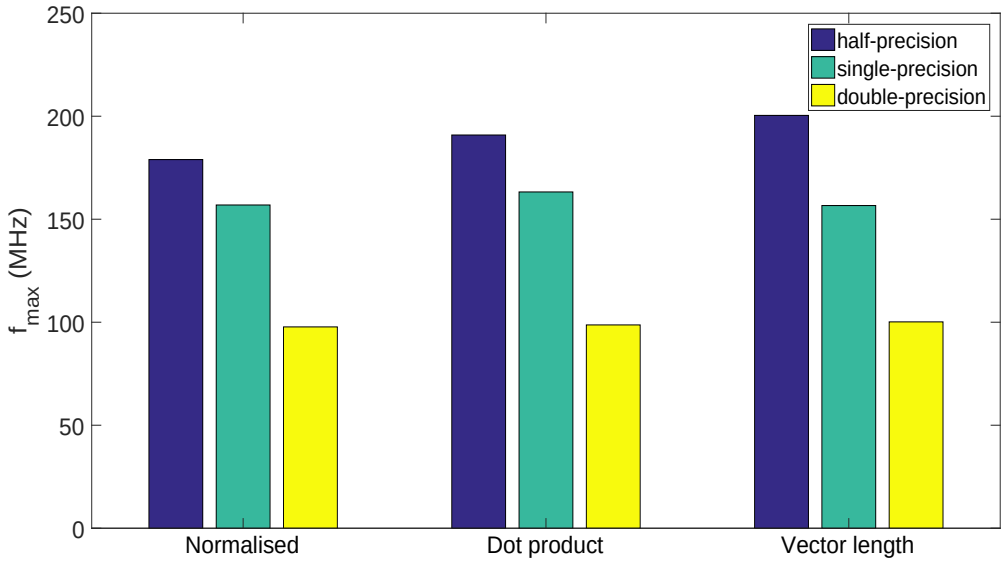


Figure 3.16: f_{max} of hardware implementations of some example vector operations.

Despite the variations in resource count between the functions, the f_{max} of each of these functions is similar for like precisions, Figure 3.16. At half-precision each of the functions can run at almost 200 MHz. At double-precision they operate at 100 MHz. Table 3.3 provides exact compilation metrics for each design at double-precision. (Appendix A provides these metrics for half- and single-precision floating-point numbers.)

The presented method for calculating matrix/vector operations works when the sizes of

the matrix remain small; for instance in the graphics rendering applications in Chapter 5. Graphics rendering typically limits matrices or vectors to a maximum of four elements in any dimension. If the matrices become too big, the system becomes I/O bound, where there are not enough pins on the device to implement the proposed schemes. In the cases of I/O bound problems other techniques must be considered to partition the problem, such as polyhedral compilation [139].

Chapter 5 will make use of these mathematical operations in an example application. This application is a graphics processor built using the FPGA fabric of an FPGA-SoC device. This off-loads the task from the processor, providing higher system throughput and efficiency.

3.4 Summary

This Chapter has focused on the implementation of fundamental floating-point mathematical operations using reconfigurable hardware. Operations considered to be fundamental are addition, multiplication, comparison and typecast. The implementations of these fundamental functions result in hardware that is accurate to 1 ULP or fewer, thus complying with IEEE-754R. The Chapter has presented resource and performance metrics for each fundamental function that has been implemented.

From these operations, matrix/vector operations are constructed. Each implementation of the matrix/vector multiply operation demonstrated optimisation for either resource use or performance. Fewer adders and multipliers reduces demand on resources. However, the adders and multipliers must be reused, leading to feedback loops with higher end-to-end latency and a lower issue rate.

Chapter 4 discusses more complicated functions, such as reciprocal, square-root and exponent. A number of methods are presented for implementing the hardware. The benefits and drawbacks of each implementation are considered. Chapter 4 also presents a case study that implements the Hodgkin-Huxley model of a neuron using different methods to approximate the exponential function. The functionality, size and performance of each neuron implementation will be analysed.

Chapter 4

Hardware Implementations of Complicated Maths Functions

In Chapter 3 a number of implementations for fundamental floating-point mathematical functions were presented. Using these implementations, methods of implementing vector and matrix operations with different types of optimisations were considered. Matrix/vector operations usually require iterating over the entire input space to create the output. Iteration leads to either an increase in resource cost or a decrease in throughput.

This Chapter will present implementations for a number of complex mathematical functions. These functions may be approximated from a combination of fundamental operations. This Chapter will demonstrate that, while iterative approximations in hardware are possible, it not always the most pragmatic approach. A number of alternative methods for implementing some functions are presented. The effect on resource cost, performance and accuracy is discussed.

This Chapter also presents a case study in which the exponential operation in the transfer function of the Hodgkin-Huxley neuron model is replaced. The Hodgkin-Huxley model is defined by sets of coupled ordinary differential equations. The dynamics are given by equations (4.1 to 4.4). This has been done before, such as the implementation presented in [140]. This used the Altera DSP builder to create a replacement that used approximations for the exponential function that are better suited to a hardware implementation. In contrast, the FPGA implementation of the Hodgkin-Huxley model presented by this Chapter will report the resource cost for each implementation.

$$C\frac{dV_i}{dt} = g_{Na}^{max} m^3 h(V_{Na} - V_i) + g_k^{max} n^4 (V_K - V_i) + g_{CL}^{max}(V_{CL} - V_i) + I_{syn}^{i,j} \tag{4.1}$$

$$\frac{dm_i}{dt} = \alpha_{mi}(1 - m_i) - \beta_{ni} n_i \tag{4.2}$$

$$\frac{dn_i}{dt} = \alpha_{ni}(1 - n_i) - \beta_{ni} n_i \tag{4.3}$$

$$\frac{dh_i}{dt} = \alpha_{hi}(1 - h_i) - \beta_{hi} h_i \tag{4.4}$$

Where V_i is the transmembrane potential of the i^{th} neuron, t is time, n_i, m_i, and h_i are the gating variables for potassium activation, sodium activation and sodium inactivation, and α and β are defined by equations (4.5 to 4.10).

$$\alpha_{ni} = \frac{0.01(V_i + 55)}{1 - exp(-(V_i + 55)/10)} \tag{4.5}$$

$$\beta_{ni} = 0.125exp(-(V_i + 65)/80) \tag{4.6}$$

$$\alpha_{mi} = \frac{0.1(V_i + 40)}{1 - exp(-(V_i + 40)/10)} \tag{4.7}$$

$$\beta_{mi} = 4exp(-(V_i + 65)/18) \tag{4.8}$$

$$\alpha_{hi} = 0.07exp(-(V_i + 65)/20) \tag{4.9}$$

$$\beta_{hi} = \frac{1}{exp(-(V_i + 35)/10) + 1} \tag{4.10}$$

Neural computing, bio-inspired methods and artificial intelligence are currently receiving a large amount of attention in research and industry due to their potential. While this may be true, replicating biological methods in hardware can be computationally expensive [141]. In addition, implementing neuron models in hardware present a network based problem; neural networks have high interconnectivity requirements. A discussion of large-scale network-on-

chips were presented in [142]. The power dissipation, latency and area of a number of network topologies was considered. This included circuit-switched networks, wormhole flow control networks, virtual channel flow control-based networks, and a speculative, single cycle, virtual channel network. Each network had different features and the choice of topology impacted the key performance metrics. Hence to create a low-cost design, the network topology must be carefully considered.

4.1 Iterative floating-point approximations and efficient hardware implementations

4.1.1 Division

In Chapter 3, implementations of addition, subtraction and multiplication were presented. The divide operation completes this set of basic mathematical operations. Implementing the divide function is more complex as it requires iteration. This Chapter presents an implementation of the divide function using Newton-Raphson's method.

Newton-Raphson's method provides a method for calculating the result of an operation based on a starting approximation. The method is not confined to calculating the inverse of a number; it can also determine the roots of any real-valued function.

Equation (4.11) is the Newton-Raphson approximation for calculating the inverse of a number using successive approximation. D is the divisor (input number) and X_i is the successive approximation of the root of the equation. X_0 has been set to 2.9142 as this has been determined to provide a good starting point for quick convergence to the answer [143] .

$$X_{i+1} = X_i - \frac{f(X_i)}{f'(X_i)} = X_i - \frac{1/X_i - D}{-1/X_i^2} = X_i(2 - DX_i) \tag{4.11}$$

Increasing the accuracy has an associated cost since the Newton-Raphson method relies on successive approximation. Table 4.1 shows resource count and performance metrics for double-precision implementations of Newton-Raphson inversion and division. The division function is implemented using a floating-point multiply after approximating the reciprocal of the divisor. (See Appendix A for information on single- and half-precision implementations).

Each iteration of the Newton-Raphson method requires two floating-point multiply and one floating-point subtraction operations. The implementation can be either performance or resource optimisation. The performance optimised version instantiates two new multipliers and a new subtraction module for every iteration. The result from one stage is fed into

Table 4.1: Resource requirements and timing analysis for floating-point invert and division operations using recursive a Newton Raphson approach to different numbers of iterations. Implementations are using double-precision floating-point accuracy.

Module	Iterations	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs Recoverable by Dense Packing	[C] Estimate of ALMs Unavailable	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
Newton-Raphson inversion	1	3178.0 (423.7)	4717.5 (506.4)	1551.5 (82.8)	12.0 (0.1)	3067 (564)	8199 (261)	8	102.42	97.89
Newton-Raphson inversion	2	5151.0 (591.9)	7632.5 (686.7)	2513.5 (94.7)	32.0 (0.0)	4939 (828)	13345 (261)	16	101.77	98.46
Newton-Raphson inversion	3	7070.0 (749.9)	10499.0 (858.1)	3464.5 (108.6)	35.5 (0.4)	6812 (1092)	18491 (261)	24	97.13	93.99
Newton-Raphson inversion	4	9047.5 (927.8)	13325.5 (1046.5)	4332.0 (119.0)	54.0 (0.3)	8682 (1356)	23637 (261)	32	96.87	93.62
Newton-Raphson inversion	5	10978.0 (1093.2)	16283.5 (1220.9)	5358.5 (128.2)	53.0 (0.5)	10551 (1620)	28783 (261)	40	95.87	93.08
Newton-Raphson inversion	6	12917.5 (1262.1)	19098.5 (1399.7)	6267.0 (141.8)	86.0 (4.2)	12422 (1884)	33929 (261)	48	100.43	96.39
Newton-Raphson inversion	7	14864.5 (1422.6)	22050.1 (1586.4)	7285.5 (164.1)	100.0 (0.2)	14293 (2148)	39075 (261)	56	96.4	93.38
Newton-Raphson inversion	8	16777.5 (1602.5)	24902.0 (1775.0)	8193.5 (173.5)	69.0 (1.1)	16164 (2412)	44221 (261)	64	98.93	96.04
Newton-Raphson inversion	9	18740.6 (1760.8)	27721.1 (1941.8)	9069.0 (181.6)	88.5 (0.6)	18035 (2676)	49367 (261)	72	97.65	94.9
Newton-Raphson inversion	10	20699.1 (1952.6)	30499.1 (2126.5)	9881.5 (174.9)	81.5 (1.0)	19916 (2940)	54513 (261)	80	97.02	93.91
Division	1	3867.0 (346.4)	5874.0 (364.2)	2020.0 (18.0)	13.0 (0.2)	3466 (523)	10531 (0)	12	97.24	94.49
Division	2	6095.0 (509.4)	9175.5 (547.5)	3098.5 (38.2)	18.0 (0.1)	5370 (787)	16717 (0)	20	95.9	92.64
Division	3	8323.0 (686.2)	12530.0 (728.4)	4235.0 (42.3)	28.0 (0.1)	7270 (1051)	22903 (0)	28	96.23	92.78
Division	4	10563.5 (850.0)	15996.5 (915.2)	5491.0 (65.5)	58.0 (0.3)	9175 (1315)	29089 (0)	36	96.3	94.24
Division	5	12769.5 (1013.3)	19226.0 (1094.8)	6503.0 (81.9)	46.5 (0.4)	11079 (1579)	35275 (0)	44	98.26	94.74
Division	6	14972.0 (1174.8)	22612.5 (1271.8)	7699.0 (97.4)	58.5 (0.4)	12978 (1843)	41461 (0)	52	97.74	94.99
Division	7	17194.5 (1348.4)	26065.0 (1434.9)	8958.5 (87.1)	88.0 (0.5)	14888 (2107)	47647 (0)	60	96.58	93.33
Division	8	19420.0 (1530.4)	29397.0 (1645.6)	10056.5 (115.8)	79.5 (0.5)	16792 (2371)	53833 (0)	68	97.39	93.66
Division	9	21636.0 (1699.2)	32584.1 (1799.6)	11020.5 (101.0)	72.5 (0.6)	18696 (2635)	60019 (0)	76	95.58	92.56
Division	10	23853.6 (1856.2)	35604.6 (1948.1)	11875.0 (92.9)	124.0 (0.9)	20602 (2899)	66205 (0)	84	93.94	90.79

the next stage, resulting in an issue rate of one (a new result is generated on every clock cycle). The resource optimisation version re-uses the existing multipliers and adders, similar to matrix/vector operations in Chapter 3. This means that a minimum number (one) of each needs to be built. The latency increases as the module cannot accept a new input while calculating the current answer. Table 4.1 shows metrics for **performance** optimised implementations. The maximum throughput of the module is equal to its f_{max}. Increasing the size of the implementation has a small impact on the f_{max}. The device has plenty of resources so each stage is implemented using a new set of logic, which does not add routing complications.

Figures 4.1 to 4.6 show the resource use and f_{max} for each implementation. As expected, each additional stage increases the resource use linearly, Figures 4.1, 4.2, 4.4 and 4.5. A double-precision implementation uses another approximately 2,000 Adaptive Logic Modules (ALMs), 2,000 Adaptive Look-Up Tables (ALUTs), and 5,000 logic registers for each stage; a half-precision implementation adds approximately 500 ALMs, 500 ALUTs, and 1,200 logic

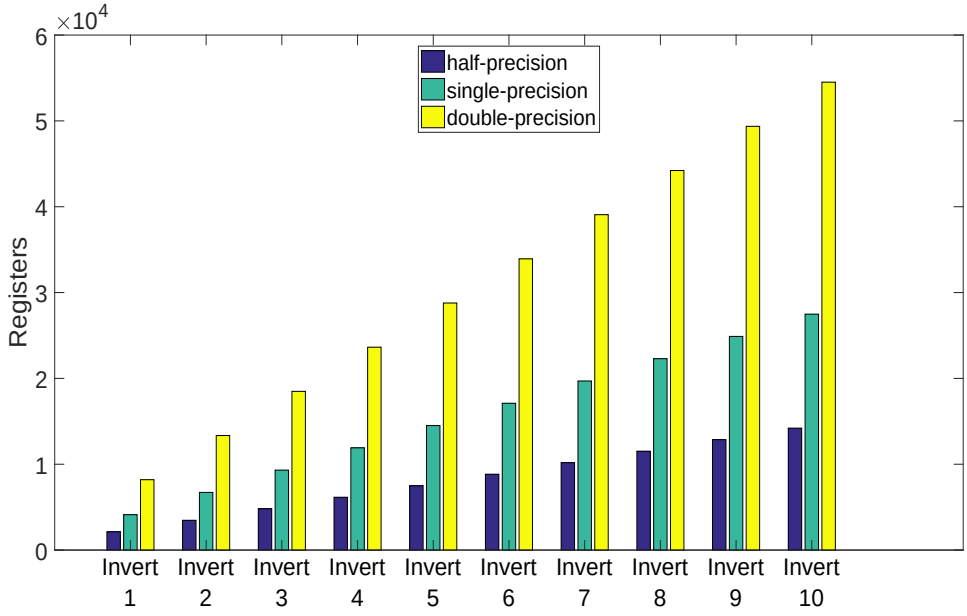


Figure 4.1: Numbers of registers required for hardware implementations of Newton-Raphson based inversion.

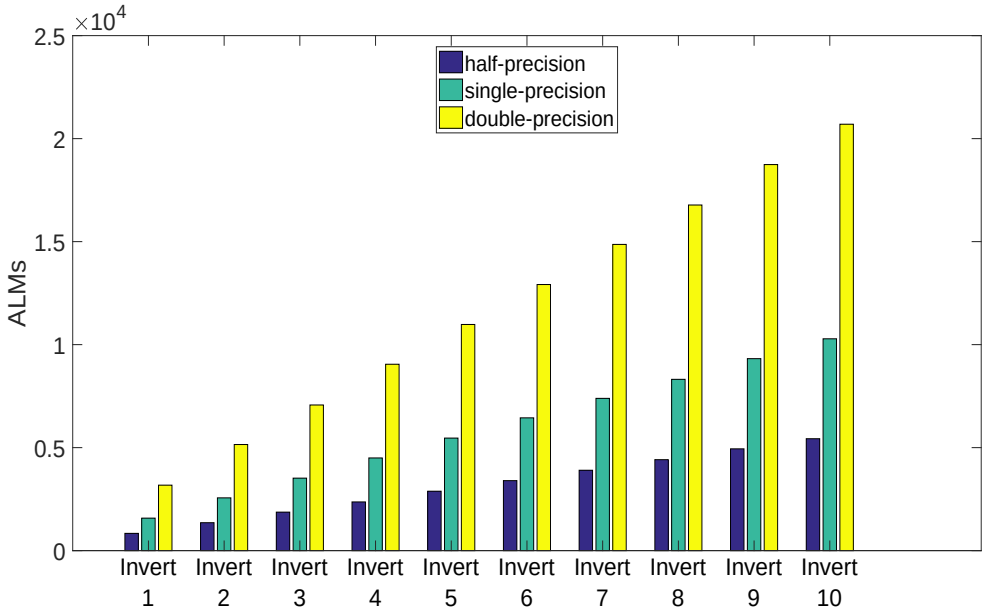


Figure 4.2: Numbers of ALMs required for hardware implementations of Newton-Raphson based inversion.

registers for each stage.

As mentioned in Chapter 3, the Intel FPGA floating-point library has an implementation for a floating-point division operation that can be implemented on a Cyclone V device. The Intel implementation is available in both single- and double-precision. The double-precision divide module can be implemented using a polynomial approximation with Newton-Raphson.

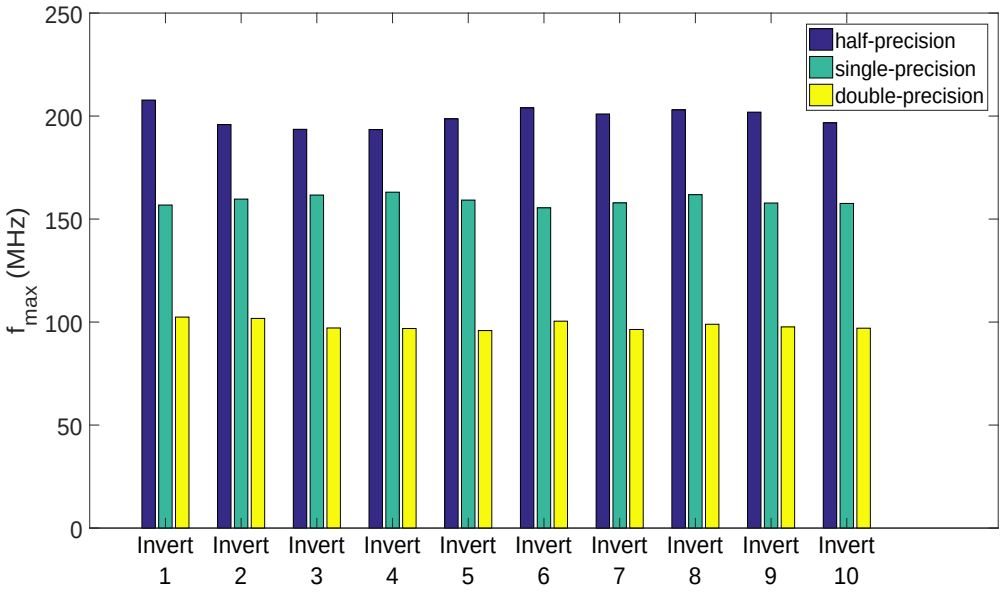


Figure 4.3: f_{max} of hardware implementation of Newton-Raphson based inversion.

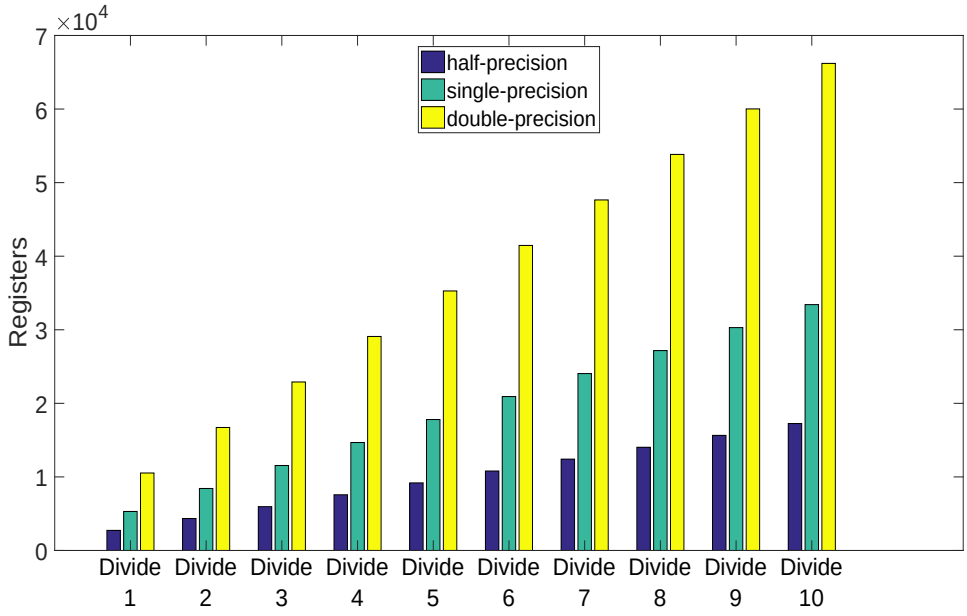


Figure 4.4: Number of registers required for hardware implementations of divide operation using Newton-Raphson inversion.

The Intel floating-point division core is a multi-cycle block that adds latency to a design. When configured for a similar f_{max} as the double-precision implementation given in this Chapter (approx. 100 MHz) the latency for the Intel core is 26 cycles. Conversely, the implementation presented by this Chapter has an issue rate of only one clock cycle. There is a resource penalty associated with the higher issue rate; the implementation provided by Intel uses only a quarter of the ALMs and almost half the DSP blocks. However, the Intel core

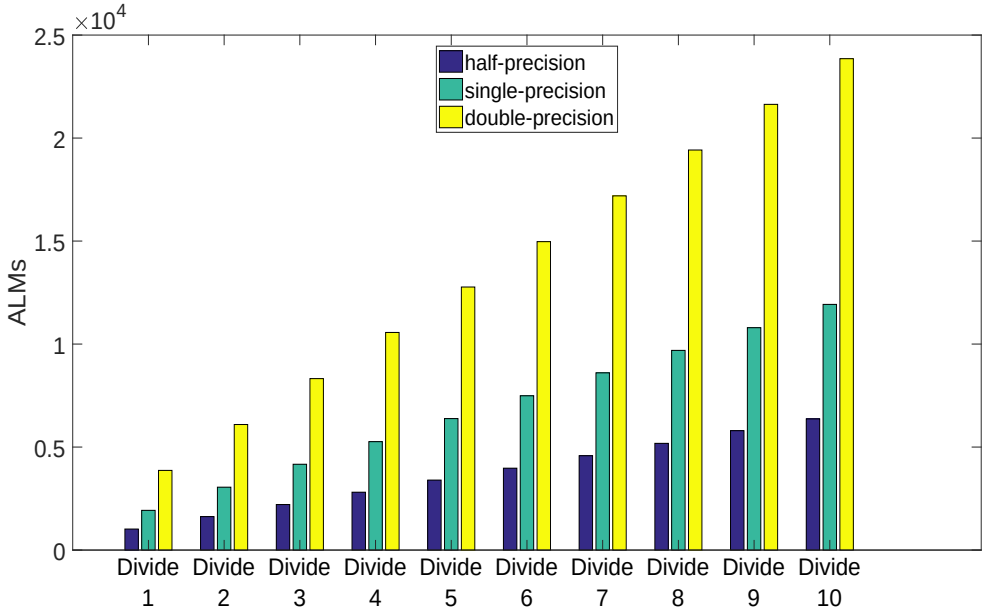


Figure 4.5: Number of ALMs required for hardware implementations of divide operation using Newton-Raphson inversion.

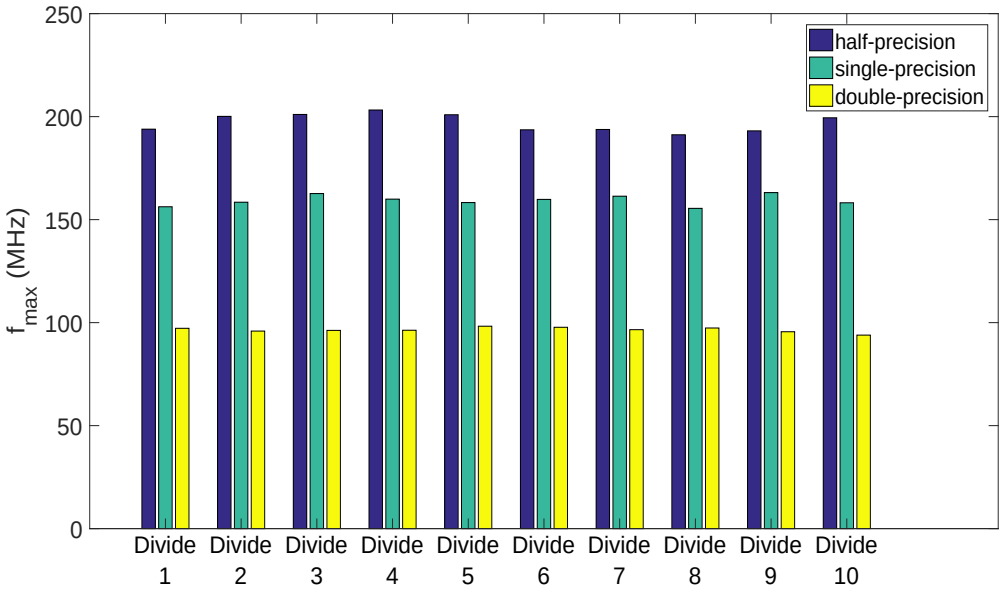


Figure 4.6: f_{max} of hardware implementations of divide operation using Newton-Raphson inversion.

requires memory blocks (389,120 bits), while the implementation provided by this Chapter does not. Similarly, the single-precision Intel divide block also introduces system latency (13 clock cycles at approx. 100 MHz) and requires memory blocks (34,303 bits). However, the Intel function requires approximately a quarter of the ALMs required for the single-precision implementation presented by this Chapter.

Figures 4.3 and 4.6 show that adding additional stages to the performance optimised implementation of the Newton-Raphson algorithm has negligible effect on the f_{max} of the system. Routing delay between logic elements tends to be the limiting factor on performance for modern Field Programmable Gate Arrays (FPGAs). There are sufficient resources on modern FPGAs that each stage of the implementation can be placed without affecting the performance of the previous stage.

Performance optimised implementations have an issue rate of one, so the maximum throughput is always equal to f_{max}. Resource optimised implementations add an additional 16 clock cycles of latency for each iteration. The throughput for resource optimisation is $f_{max}/latency$.

4.1.2 Analysis of error

It has been shown that adding more iterations is costly in terms of either resources or throughput. Convergence for each implementation can be seen from plots of error with increasing numbers of iterations.

Consider the double-precision implementation. From Figure 4.7 the relative error (Units of Least Precision - ULPs) after a single iteration is on the order of 10^{13}. This level of error is too high for the implementation to be considered useful. Additionally, the normalised error is shown to be one in some cases, meaning the result is 100% inaccurate.

Adding one more stage of the Newton-Raphson approximation to the implementation reduces the error in the result by two orders of magnitude. The relative error is still too large by the definition of acceptable error in a floating-point function ($error \leq one\,ULP$) for IEEE-754R. However, the normalised error is now in the order of 10^{-4}. Therefore, the mathematical value of the result is close to the 'true' answer.

After three iterations the normalised error is always below 10^{-9}. Depending on the application this error may not be problematic. Some applications may have other processes, functions or constraints that add quantisation and therefore errors to the process. This can eliminate the need for a mathematical result to have a relative error of less than, or equal to one. Instead, area (resource use) is more important. In this situation it is sensible to use a function with a low normalised error (but a high relative error) to save on resources. Other applications will be more sensitive to relative error, but may not have constraints such as area. In these cases greater numbers of iterations can be used.

Figures 4.8 and 4.9 plot the error in the inversion function with five iterations and ten

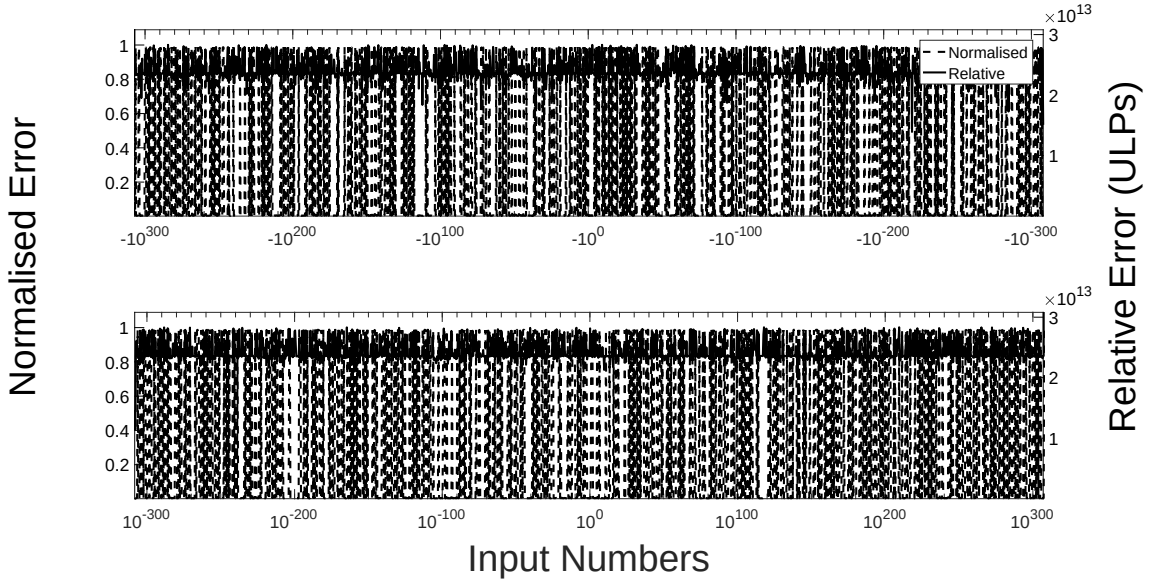


Figure 4.7: A Newton-Raphson iteration-based inverter realised in hardware with only a single NR stage. Top half of the graph shows error for a negative input, bottom half shows error for a positive input.

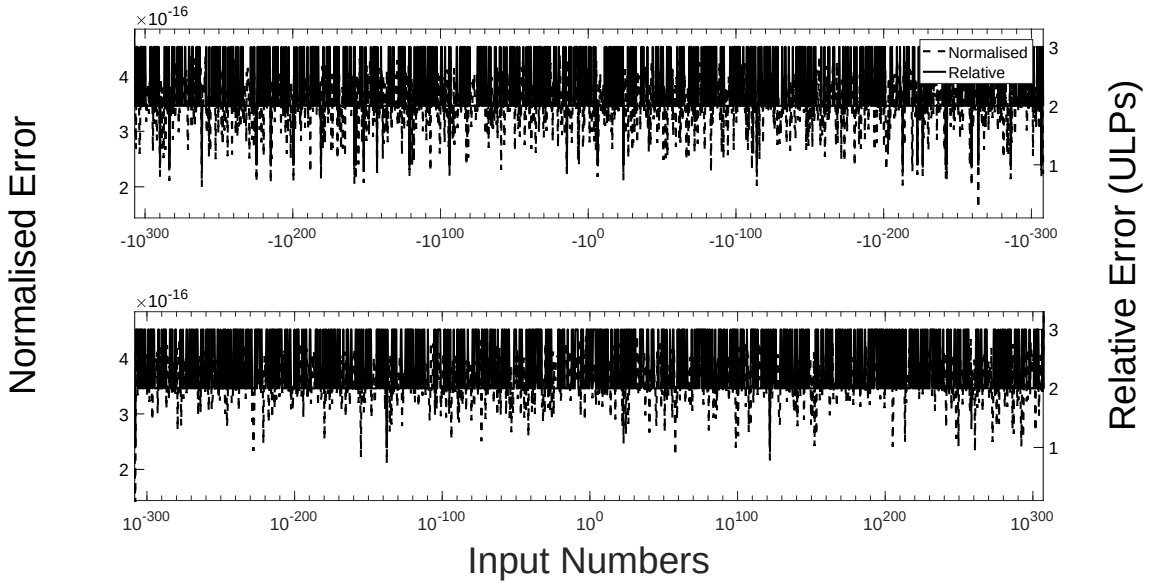


Figure 4.8: A Newton-Raphson iteration-based inverter realised in hardware with five NR stages. Top half of the graph shows error for a negative input, bottom half shows error for a positive input.

iterations respectively. The algorithm has converged - performing further iterations does not change the answer and therefore the error - and has a maximum relative error of three ULPs. (Single- and half-precision impelmentations are shown in Appendix C.) For single-precision this convergence occurs after four iterations. The half-precision implementations of the Newton-Raphson inversion shows convergence after three or fewer iterations. Due to the

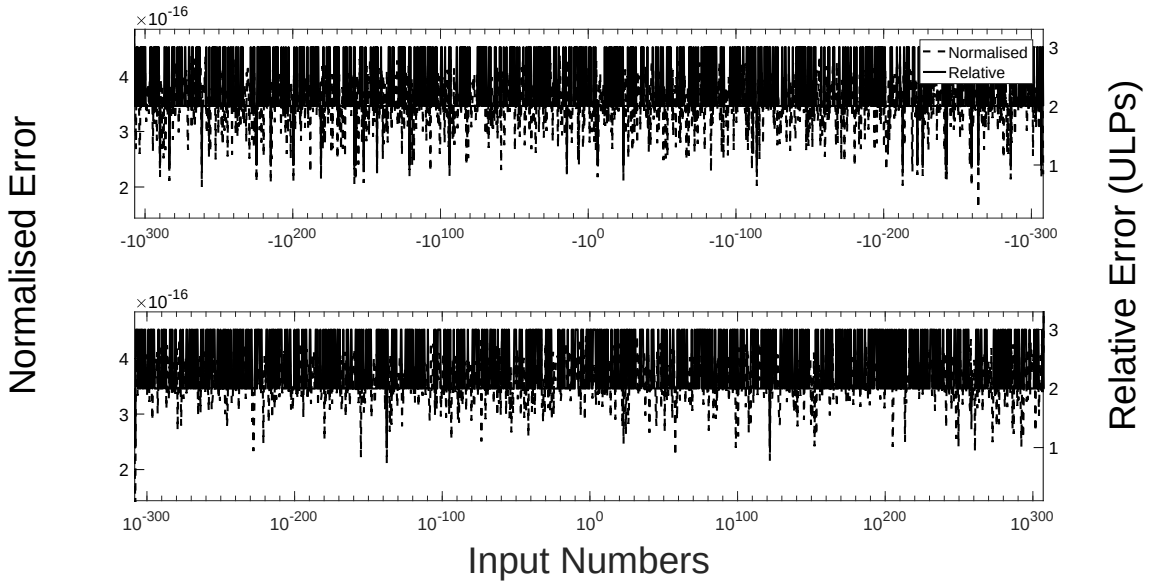


Figure 4.9: A Newton-Raphson iteration-based inverter realised in hardware with ten NR stages. Top half of the graph shows error for a negative input, bottom half shows error for a positive input.

reduction in accuracy of the half-precision number system, there are some anomalies. Inputs that cause output values that are very close to the boundary of two different exponent values can show a large amount of relative error. Despite this the normalised error for these results is low for all results, in the order of 10^{-3} or better. Depending on the application, this is not a problem.

For a relative error of three or fewer ULPs there is a large resource cost. The double-precision implementation needs almost 11,000 ALMs and ALUTs, and 29,000 logic registers. As before, reducing the floating-point precision, reduces the cost (Appendix A).

4.1.3 Square-root

Note: the research presented in this section has been accepted for publication in IET Computers & Digital Techniques.

The implementation of the Newton-Raphson method has highlighted a number of problems. These same problems will occur in any other successive approximation algorithm. These are used to evaluate complex, but common, mathematical functions such as square-root and exponential functions. Successive approximation algorithms can result in large designs, have a low throughput and suffer from compound errors. However, there are alternative methods that can mitigate these problems. This Section presents some ways of approximating a square-root.

The square-root function is ubiquitous in modern computing, for applications such as Computer Aided Design (CAD), graphics rendering and Artificial Intelligence (AI). The square-root is calculated using methods such as Newton-Raphson [54], Taylor-series expansion [53] or Goldschmidt's expansions [53]. These techniques exhibit similar problems to the reciprocal operations discussed previously in this Chapter.

There have been attempts to optimise successive approximation methods for hardware. Wang & Schulte [54] proposed finding the inverse of the square-root of x using equation (4.12). X is the number to be square-rooted and R_i is the iterative approximation of the result.

$$R_{i+1} = \frac{R_i}{3}(3 - X.R_i^2) \tag{4.12}$$

Although more optimised for maximum latency (210 clock cycles) and critical path (0.95 ns), than implementations that approximate the square-root, this still exhibits many of the original problems for implementing successive approximation in hardware.

Methods such as non-restoring algorithms and Dwandwa Yoga [56] are appealing due to their low resource count and low latency. In particular there are non-restoring methods that operate using a series of bit-wise operations, compares and shifts, which have a very low resource cost. *Non-restoring* algorithms use equation (4.13). D is the input, Q is the quotient and R is the remainder. Unfortunately, the non-restoring algorithms have a very high relative and normalised error. Adaptations will be presented that reduce the relative error in the hardware implementation to **one ULP or fewer** as per IEEE-754R.

$$D = Q^2 + R \tag{4.13}$$

```
1.  if(D_WIDTH%2 != 0)  D={0,D},
    else                D = D,
2.  Always              Q_0 = 0, F_0 = 0,
3.  Always              t = 0, i = D_WIDTH,
4.  Always              R_t = D_{i:i-1},
5.  iterate from i = D_WIDTH to 0,
6.  if((F_t << 1)|1 < R_t),
7.      if((F_t << 1)|1 > R_t)   Q_{t+1} = (Q_t<<1)|0,
                                 F_{t+1} = ((F_t + F_t[0])<<1)|0,
        else                     Q_{t+1} = (Q_t<<1)|1,
                                 F_{t+1} = ((F_t + F_t[0])<<1)|1,
    else                         Q_{t+1} = (Q_t<<1)|0,
                                 F_{t+1} = ((F_t + F_t[0])<<1)|0,
8.  Always
      R_{t+1} = (R_t - (F_{t+1} × F_{t+1}[0]))<<2|D_{i-2:i-3},
9.  Always              t = t + 2, i = i - 2,
10.     Repeat steps 6 to 8 until i = 0
```

Listing 4.1: Algorithm for calculating the square-root of a number using a non-restorative method

The principle of operation for the non-restorative algorithm is given in Figure 4.10 and Listing 4.1. D is the input number (radicand), $D_{i:i-1}$ represents a sub-group of the radicand, F_t is the partial factor, R_t is the partial remainder, Q_t is the quotient, t is the time step, and i is the bit-index. The input number is divided into sub-groups of two bits, which are parsed from the Most Significant Bit (MSB) pair to the Least Significant Bit (LSB) pair.

Due to the nature of the non-restorative algorithm, converting each sub-group of bits into a single bit for the quotient, there is a loss of accuracy. A mantissa of $n+1$ bits - n is the number of bits in the mantissa, $n+1$ represents the true mantissa - will result in a quotient of $n/2$ bits. To increase accuracy, the mantissa is padded with zeros until it is $2(n+1)$ bits in length. This is similar to a performing a successive approximation algorithm until every bit of the mantissa is calculated.

Padding the mantissa to increase the precision of the non-restoring algorithm has trade-offs. The improvements to the non-restorative algorithm can increase either the resource count or the latency. Similar to the optimisation techniques proposed in Chapter 3, the non-restorative algorithm can be either resource or performance optimised. If the resource optimised implementation is considered, the resource count remains close to the original

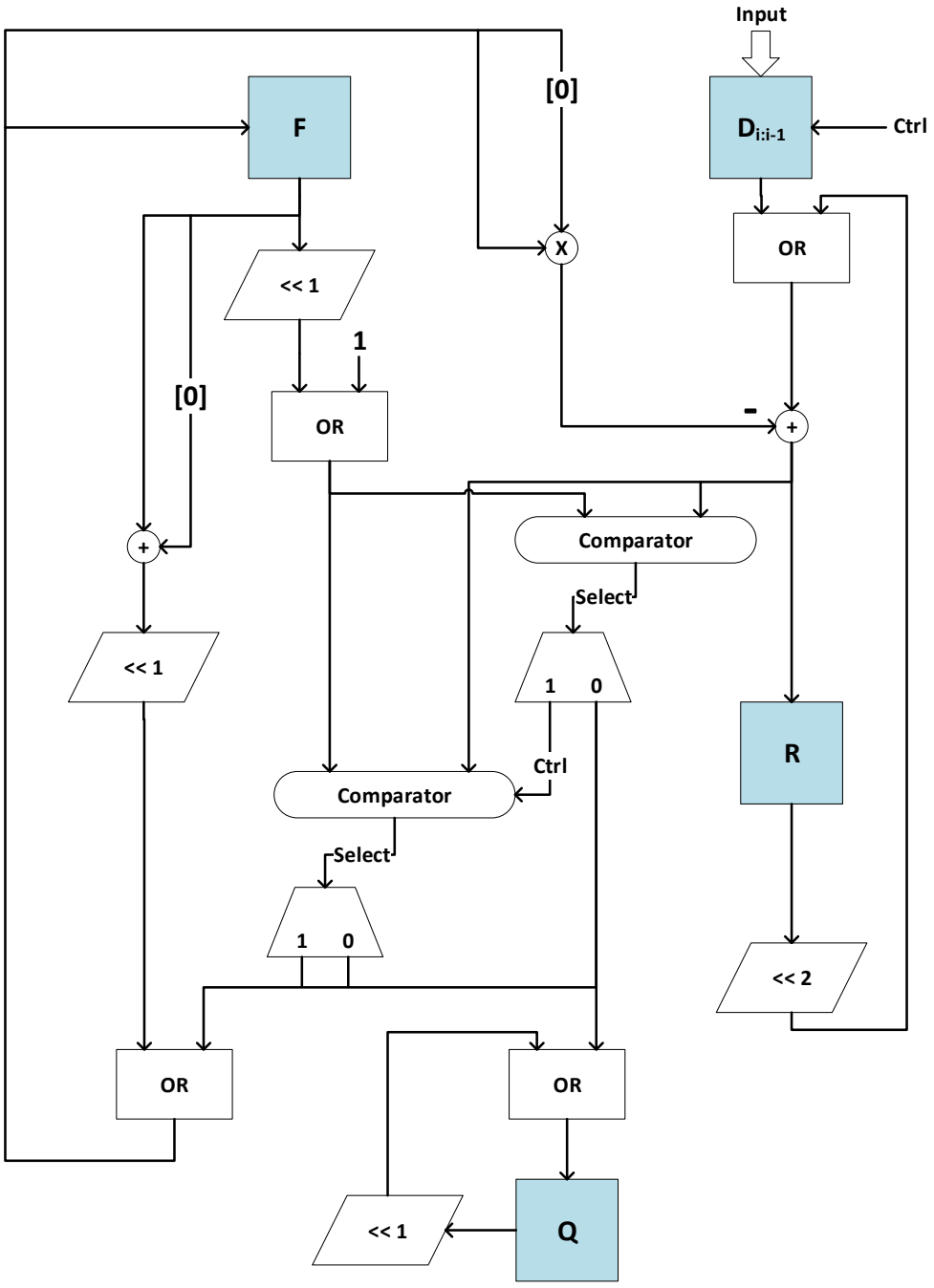


Figure 4.10: The flow of the non-restorative algorithm can be expressed using a flow chart, from which the required hardware operations and resources are derived.

algorithm. There is a slight overhead as the size of some registers must be increased to accommodate the increase in precision. However, the latency (τ) is increased to $2(n+1)+1$ clock cycles from $(n+1)+1$ for an odd length mantissa and $(n+1)+2$ for an even length mantissa.

If the performance optimised implementation is considered the issue rate remains as one. The setup latency for the module is increased. Adding pipelining stages allows the design to be clocked faster, thus increasing the overall throughput. However, each pipelining stage

adds additional latency (N_p).

If the pipeline optimisations are applied to a traditional implementation of a non-restorative algorithm, the latency is given by equation (4.14)

$$\tau = (n+1) + \frac{n}{2} \times N_p \tag{4.14}$$

for a mantissa with an odd number of bits, or equation (4.15)

$$\tau = (n) + \frac{n-1}{2} \times N_p \tag{4.15}$$

for a mantissa with an even number of bits. The set-up latency for the improved algorithm rises to equation (4.16)

$$\tau = (2(n+1)+1) + (n+1) \times N_p \tag{4.16}$$

for an odd length mantissa or equation (4.17)

$$\tau = (2(n+1)) + n \times N_p \tag{4.17}$$

for an even length mantissa. Latencies for all implementations are summarised in Table 4.2.

Table 4.2: Latency calculations for resource and performance optimised implementations of the non-restorative square-root module. Latency (τ) is given in clock cycles.

		Traditional algorithm	**Improved accuracy algorithm**
Resource optimised	Odd mantissa	$\tau = (n+1)+1$	$\tau = 2(n+1)+1$
	Even mantissa	$\tau = (n+1)+2$	$\tau = 2(n+1)+1$
Performance optimised	Odd mantissa	$\tau = (n+1)+((n/2) \times N_p)$	$\tau = (2(n+1)+1)+((n+1) \times N_p)$
	Even mantissa	$\tau = (n)+(((n-1)/2) \times N_p)$	$\tau = (2(n+1))+(n \times N_p)$

The resource costs and performances of all non-restoring square-root designs have been considered, shown in Figures 4.11 to 4.13. All designs were implemented in half-, single- and double-precision, using both pipelined and non-pipelined methods. For pipelined designs, data is shown for five pipeline stages.

Adding pipeline stages causes a significant increase in the resource use of both the traditional and new non-restorative algorithms, Figures 4.11 and 4.12. At double-precision the increase in resource cost compared to half-, or single-precision is large. Each pipeline stage is used to register data through D-type flip-flops for the next stage. When the module is configured for double-precision, the widths of the registers used at each pipeline stage is considerably longer than at half-, or single- precision, leading to a significant resource increase.

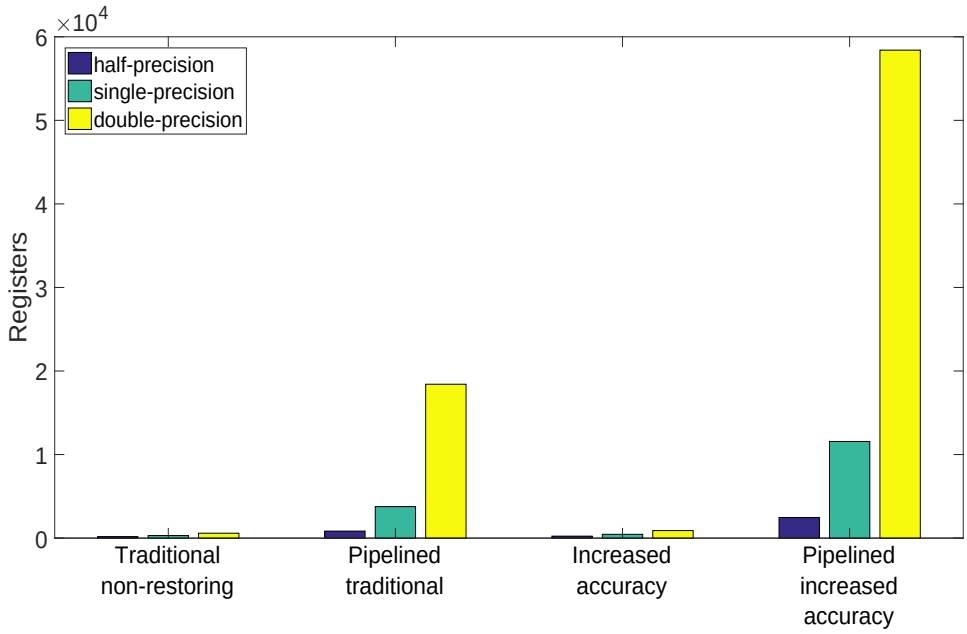


Figure 4.11: Registers required for hardware implementations of square-root operation.

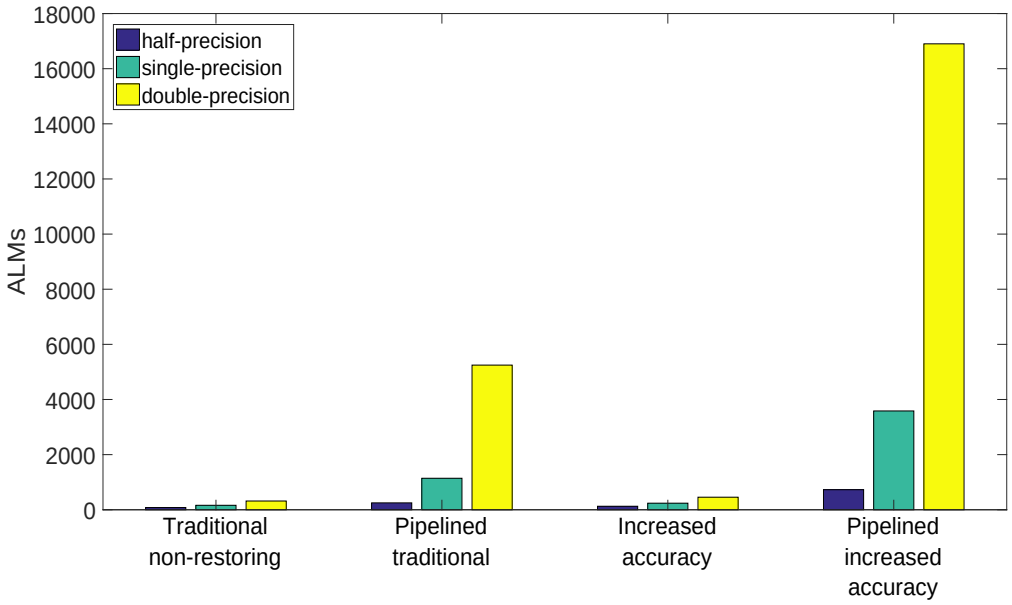


Figure 4.12: ALMs required for hardware implementations of square-root operation.

Table 4.3 shows exact numbers for the resource cost at double-precision. (Appendix A shows figures for half- and single-precision).

Figure 4.13 shows that when the module is arranged in a pipelined version of either the traditional or the new non-restoring system, the overall f_{max} is increased over the respective non-pipelined version. Pipelining stages allow the fitter to reduce routing delay, which allows a faster clock to be used.

Increasing the accuracy with the new algorithm increases the number of resources re-

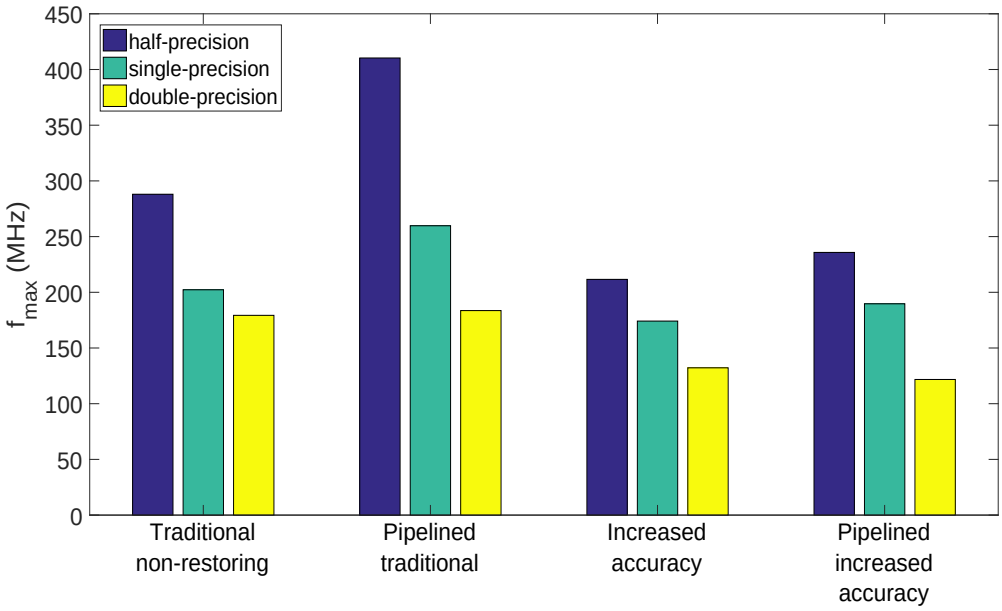


Figure 4.13: f_{max} of hardware implementations of square-root operation.

quired. The more resources that are required, the more complicated routing a design can require. Even with pipelining stages, there is a decrease in f_{max}.

Importantly the additions to the algorithm, although at a cost, have reduced the error to no more than a single ULP, shown in Figures 4.14 to 4.16. This also translates to an 1.36×10^8 fold decrease in the normalised error for double-precision.

Table 4.3: Resource requirements and timing analysis for hardware friendly implemenatations of a floating point square root operation. Implementations are using double-precision floating-point accuracy.

Module	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs Recoverable by Dense Packing	[C] Estimate of ALMs Unavailable	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
Traditional without pipeline	317.5 (317.5)	388.0 (388.0)	72.0 (72.0)	1.5 (1.5)	436 (436)	583 (583)	0	179.31	180.67
Traditional with pipeline	5247.5 (5247.5)	9149.1 (9149.1)	3984.6 (3984.6)	83.0 (83.0)	2288 (2288)	18416 (18416)	0	183.59	185.91
Proposed new design without pipeline	455.5 (455.5)	577.5 (577.5)	126.5 (126.5)	4.5 (4.5)	589 (589)	901 (901)	0	132.21	133.92
Proposed new design with pipeline	16901.9 (16901.9)	29184.5 (29184.5)	12486.6 (12486.6)	204.0 (204.0)	8718 (8718)	58404 (58404)	0	121.74	124.29

FPGA vendors may provide Intellectual Property (IP) blocks for functions such as the square-root of a floating-point number. Metrics for IP provided by Intel are given in Table 4.4. Intel provide different IP for the Cyclone V and Stratix V range of devices, both of which have been synthesised at single- and double-precision. The latency for the Cyclone V IP is 16 clock cycles at single-precision and 30 clock cycles at double-precision. Metrics for the

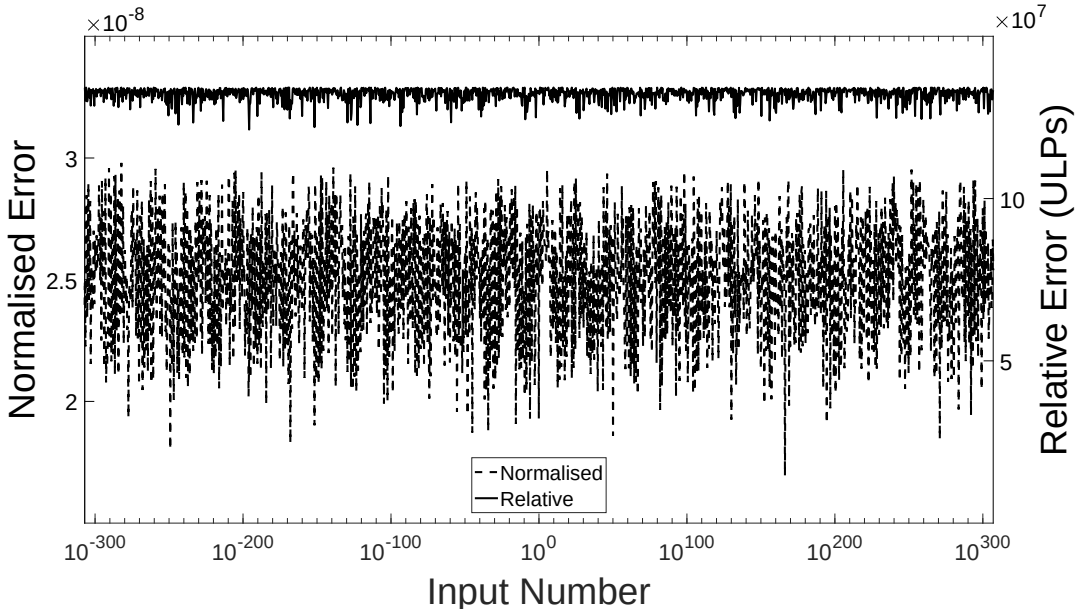


Figure 4.14: Traditional non-restorative algorithms can only calculate a mantissa that is half the width of the input. This creates significant error in the output, as shown with the double-precision system here.

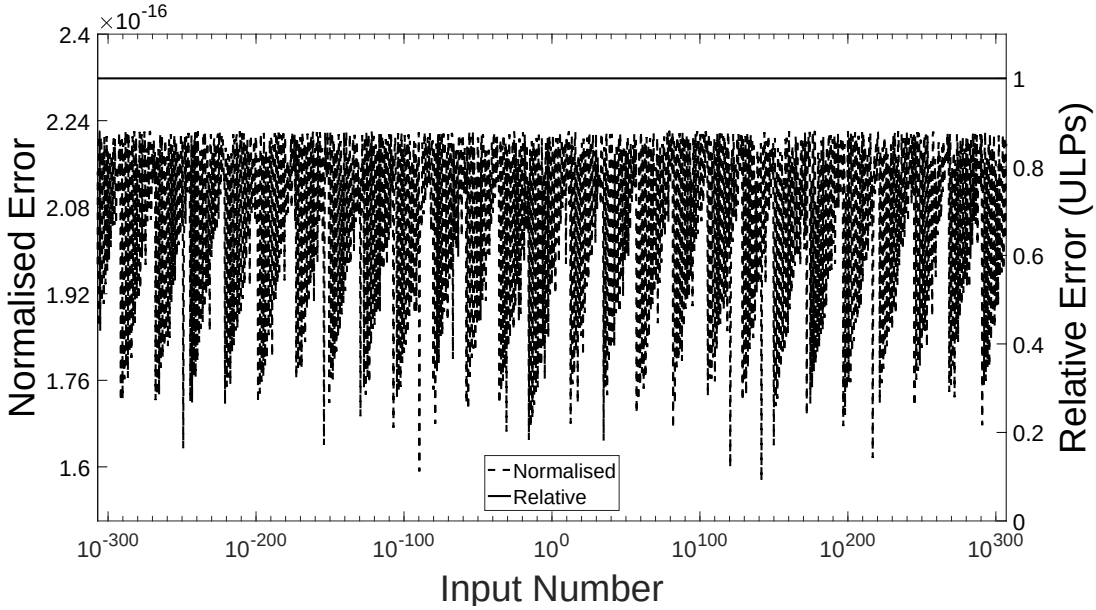


Figure 4.15: The amendments to the algorithm allow all the bits of the mantissa to be calculated, drastically increasing the accuracy to a worst case of one ULP.

Cyclone V device come from the compilation report from Quartus II. For the high performing Stratix V device the metrics may be found in [144].

The Intel IP does not provide the issue rate of one achievable by the performance optimised non-restoring implementation. Additionally, for the high performance Stratix V device, the megafunction uses Digital Signal Processing (DSP) blocks. DSP blocks are a valuable resource that may be better used elsewhere in a design. Comparison can be drawn between the

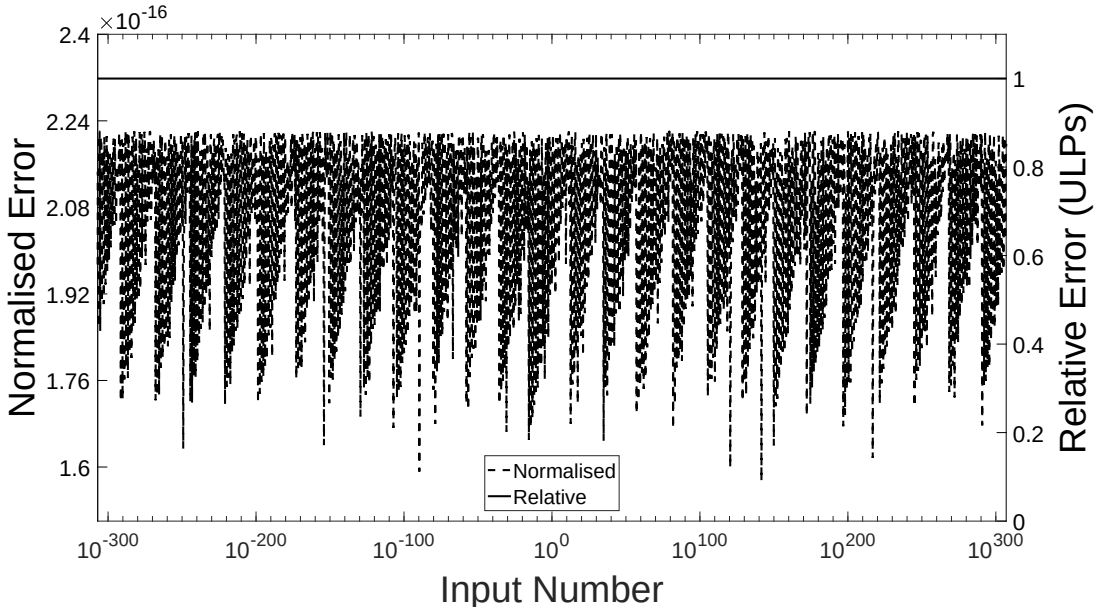


Figure 4.16: Adding pipelining to the square-root calculation algorithm is not detrimental effect to the accuracy of the output.

ALMs used for the Intel FPGA square-root function and the square-root function proposed in this Chapter, both of which targeted Cyclone V technology. Without pipelining both implementations from this Chapter require fewer ALMs than the Intel square-root cores for both single- and double-precision implementations. The addition of pipelining significantly increases the resource consumption of the proposed method, but it does achieve a much higher throughput that the Intel cores.

Table 4.4: Intel provide a megafunction core to perform the square-root operation on floating-point numbers. The resources used and performance vary depending on the width of the floating-point number and the target device.

		Single-Precision	**Double-Precision**
Cyclone V	f_{max} (MHz)	135.94	104.88
	Throughput (MFLOPs)	8.50	3.50
	ALMs	192	888
	Registers	396	1783
	DSP	0	0
Stratix V	f_{max} (MHz)	393.7	274.12
	Throughput (MFLOPs)	65	16
	ALMs	112	458
	Registers	136	1060
	DSP	2	9

Non-restoring algorithms allow the square-root of a number to be calculated using very few

resources compared to successive approximation techniques, importantly the implementation requires no DSPs. The use of DSP blocks in the IP cores from Intel FPGA results in smaller implementations compared to the pipelined non-restoring method. However, the issue rate of one allows the non-restoring implementation to have a much higher throughput. Traditional methods for implementing the non-restorative method lead to a large relative error in the output. Additions have been made to the algorithm that reduce the relative error to no more than one ULP. This is in line with the maximum allowable error for floating-point functions. (Additional plots of error for double-, single- and half-precision implementations can be seen in Appendix D.)

4.1.4 Exponential

Evaluating the exponential function presents similar problems as the reciprocal or square-root. There are a number of mathematical methods for calculating exponents that rely on successive approximation. These are often bounded approximations designed for small inputs, or require a large number of iterations to be accurate. Two common methods are the Euler expansion of continued fraction approximation, equation (4.18), and the power series approximation, equation (4.19).

$$e^x = 1 + \frac{x}{1 - \frac{x}{x+2-\frac{2x}{x+3-\frac{3x}{x+4-\ldots}}}} \tag{4.18}$$

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + ... \tag{4.19}$$

From the equations, it can be seen that these approximations exhibit similar problems to the Newton-Raphson methods. A large number of operations are required, and each operation is potentially costly (particularly division, Table 4.1). Identifying other methods for approximating the exponent of a number is therefore necessary for hardware implementation.

It is possible to rewrite the formula $R = e^x$ in the form of $R = 2^a$. This is more computationally friendly for a binary processor. In the rearrangement a is scaled version of x using a factor of $ln(2)$, for example $R = 2^{x/ln(2)}$. This is similar to the method used in the *math.h* C library [145] where the $exp(x)$ function uses the relationship $2^{xln(2)}$.

4.1.4.1 Implementing the exponent function on a processor

It is possible to implement these approximations on a processor, but achieving high accuracy is costly in terms of clock cycles. Processors actually use an alternative method, Chebyshev approximation, without need for a large number of iterations - *math.h* [145]. The equivalent function reduces the $exp(x)$ to the form shown in equation (4.20) where $|r| \leq 0.5ln(2) \approx 0.34658$.

$$x = k \times ln(2) + r \tag{4.20}$$

Where the $exp(r)$ can be calculated using a bounded approximation for a small input value, equation (4.21).

$$\exp(r) = 1 + \frac{2r}{R - r} = 1 + r + \frac{r \times R1(1)}{2 - R1(r)} \tag{4.21}$$

R is approximated as a fifth order polynomial.

From equations (4.20) and (4.21), it is concluded that the exponent can be approximated as in equation (4.22).

$$\exp(x) = 2^k + exp(r) \tag{4.22}$$

The processor implementation uses both an approximation in the form 2^a and a bounded approximation for a small input value, equation (4.21), [145].

On a processor this method of approximating the exponent function leads to a highly accurate answer for relatively little processor cost. However, due to evaluating a fifth order polynomial and the division operation for the $exp(r)$ approximation, implementing equation (4.22) in hardware is costly. Other approaches for estimating the exponential of a number using hardware will now be presented.

4.1.4.2 Hardware implementations of the Euler and power series methods

It has already been proposed that the Euler and power series approximation methods are impractical for implementation both on a processor and in hardware, particularly when efficiency or area are key criteria. However, for comparison the Euler and power series approximations have been implemented in hardware. Tables 4.5 and 4.6 show the resources required for Euler and power series methods using half-precision floating-point. Due to both methods requiring division, two versions of each implementation have been synthesised. One version

uses a single stage Newton-Raphson division and the other uses five stages. The single stage division implementation requires fewer resources but has higher error. Five stages ensures the division result has converged; the effect of convergence of the division operation in the error of the exponential function can then be analysed. (Synthesis data for double-, single- and half-precision is in Appendix E.)

Table 4.5: Resource requirements and timing analysis for Euler and power series mathematical expansions of e^x. Implementations are using half-precision floating-point accuracy, using a five stage Newton-Raphson inversion.

Module	Iterations	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs recoverable by Dense Packing	[C] Estimate of ALMs unavailable	Combinational ALUTs	Dedicated logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
euler	1	8789.0 (237.4)	13091.0 (257.7)	4359.5 (20.3)	57.5 (0.0)	7418 (383)	24728 (0)	24	192.46	192.75
euler	2	13807.5 (333.2)	20384.5 (352.7)	6671.0 (19.5)	94.0 (0.0)	11015 (545)	39709 (0)	35	193.91	193.87
euler	3	18757.0 (444.8)	27832.0 (481.4)	9188.0 (36.7)	113.0 (0.0)	14477 (721)	54786 (0)	46	192.94	189.9
euler	4	23841.5 (548.3)	35173.0 (575.2)	11517.5 (26.9)	186.0 (0.0)	18045 (884)	69863 (0)	57	195.2	193.65
euler	5	28486.5 (556.9)	34537.0 (605.1)	6133.0 (48.6)	82.5 (0.3)	21567 (1049)	84940 (0)	68	187.06	185.8
euler	6	33543.0 (639.4)	38554.0 (670.4)	5132.5 (31.0)	121.5 (0.0)	25069 (1225)	100017 (0)	79	174.19	170.15
power	1	4020.0 (132.4)	5893.0 (135.1)	1919.0 (2.7)	46.0 (0.0)	3685 (204)	10996 (17)	12	207.3	201.17
power	2	8209.5 (228.2)	12124.0 (252.9)	3983.0 (24.7)	68.5 (0.0)	7116 (360)	22783 (17)	24	191.68	187.06
power	3	12368.0 (329.3)	18186.0 (357.1)	5880.0 (27.8)	62.0 (0.0)	10528 (529)	34671 (17)	36	190.48	185.01
power	4	16575.0 (434.3)	24648.0 (468.7)	8161.0 (34.4)	88.0 (0.0)	14106 (697)	46557 (17)	48	198.26	196.08
power	5	20751.5 (531.9)	30338.5 (572.9)	9714.0 (41.0)	127.0 (0.0)	17567 (866)	58443 (17)	60	181.82	183.99
power	6	24982.0 (635.8)	36128.0 (670.1)	11267.0 (34.2)	121.0 (0.0)	21236 (1031)	70331 (17)	72	186.05	192.09

Table 4.6 shows a very high resource cost for an implementation that uses a resource-optimised division stage. The size of each implementation grows rapidly as more iterations are introduced. At half-precision floating-point each additional Euler iteration requires approximately 1800 ALMs, 5300 registers and three DSP blocks; each power series iteration requires approximately 1500 ALMs, 4185 registers and four DSP blocks. Similar to methods discussed in Chapter 3, these implementations can be arranged to reuse the same set of logic multiple times, calculating the result to greater accuracy. The effect would be a decrease in throughput.

Methods that use successive approximation suffer from truncation error. The more terms that are truncated, the larger the error can be. The error can be expressed mathematically using $y(t)$ to denote the exact solution, $y(t_j)$ to denote the exact solution at step j, and y_j

Table 4.6: Resource requirements and timing analysis for Euler and power series mathematical expansions of e^x. Implementations are using half-precision floating-point accuracy, using a single stage Newton-Raphson inversion.

Module	Iterations	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs recoverable by Dense Packing	[C] Estimate of ALMs unavailable	Combinational ALUTs	Dedicated logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
euler	1	3544.5 (126.2)	5137.0 (131.4)	1603.0 (5.2)	10.5 (0.0)	3293 (195)	9640 (0)	8	191.68	191.35
euler	2	5343.5 (176.6)	7900.5 (187.7)	2576.0 (11.1)	19.0 (0.0)	4617 (266)	14965 (0)	11	200.48	196.08
euler	3	7193.0 (207.1)	10775.0 (228.6)	3631.5 (21.5)	49.5 (0.0)	5932 (337)	20322 (0)	14	189.79	195.2
euler	4	8977.5 (250.2)	13540.0 (269.9)	4631.0 (19.8)	68.5 (0.0)	7266 (409)	25679 (0)	17	195.69	193.05
euler	5	10801.5 (291.9)	16036.0 (321.9)	5291.5 (30.0)	57.0 (0.0)	8653 (475)	31036 (0)	20	192.2	186.81
euler	6	12659.5 (340.9)	18795.0 (368.7)	6247.5 (27.8)	112.0 (0.0)	9997 (552)	36393 (0)	23	189.93	192.09
power	1	1712.0 (71.6)	2453.5 (74.3)	752.0 (2.8)	10.5 (0.0)	1628 (98)	4406 (17)	4	176.18	174.58
power	2	3169.5 (115.8)	4650.0 (123.2)	1498.0 (7.4)	17.5 (0.0)	2938 (175)	8591 (17)	8	206.83	200.4
power	3	4705.0 (164.8)	6886.0 (174.2)	2205.0 (9.4)	24.0 (0.0)	4277 (239)	12807 (17)	12	202.43	201.05
power	4	6196.0 (199.5)	9069.0 (218.7)	2905.5 (20.0)	32.5 (0.9)	5603 (315)	17021 (17)	16	199.4	194.48
power	5	7791.0 (247.3)	11399.0 (266.2)	3659.5 (18.9)	51.5 (0.0)	6997 (390)	21235 (17)	20	205.25	199.28
power	6	9325.5 (286.3)	13651.0 (313.4)	4385.5 (27.1)	60.0 (0.0)	8413 (458)	25451 (17)	24	196.89	193.39

to denote the numerical solution at j. Therefore the error is given by equation (4.23)

$$e_j = |y(t_j) - y_j| \tag{4.23}$$

Increasing the number of iterations of either the Euler or power series method will increase the accuracy of the result. However, in order to get an accurate result for large input values, the number of iterations required tends to infinity. Figures 4.17 to 4.20 demonstrate the error in the result of both Euler and power series methods over the representable range of half-precision floating-point after ten iterations. Figures 4.17 and 4.18 use a five stage Newton-Raphson division implementation compared to the single stage used in Figures 4.19 and 4.20. It can be seen from the Figures that increasing the accuracy of the division stage has little effect on the accuracy of the exponential approximation due to the inaccuracy incurred using only ten iterations of the algorithm. It can be concluded that both Euler expansion of continued fraction and power series approximations are inappropriate for hardware implementation.

4.1.5 Hardware implementations of curve-fitting methods

Section 4.1.4 showed that $exp(x) = 2^a$, where a is x scaled by a factor $ln(2)$. Implementing 2^a is far more resource-friendly. To increase the accuracy of the result while using a small number

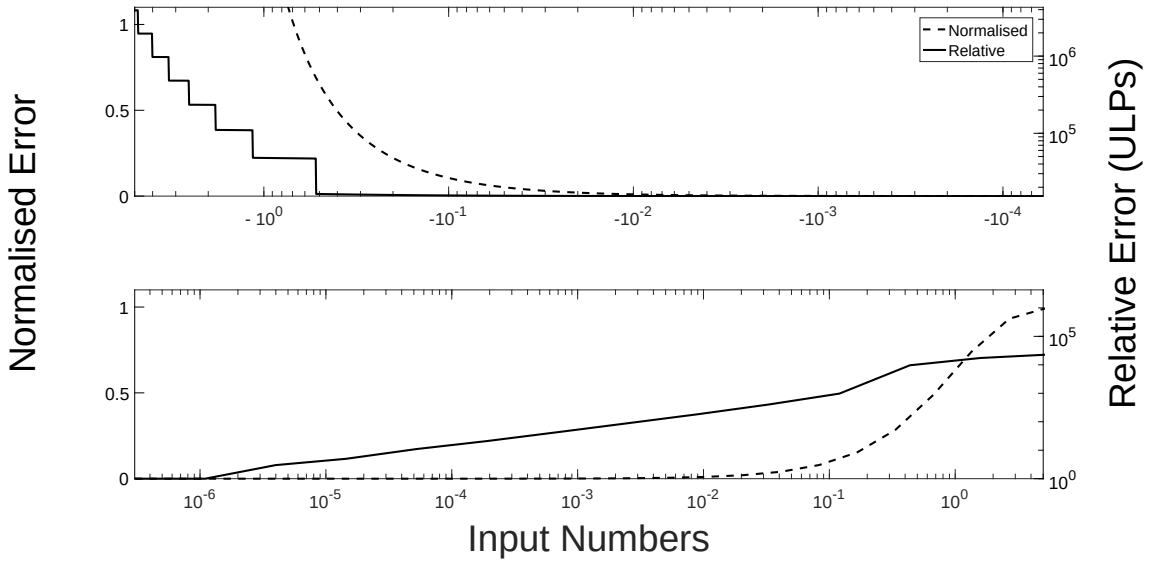


Figure 4.17: Euler series approximation to 10 iterations in half-precision using a five stage Newton-Raphson division implementation. Top half of the graph shows error for a negative input, bottom half shows error for a positive input.

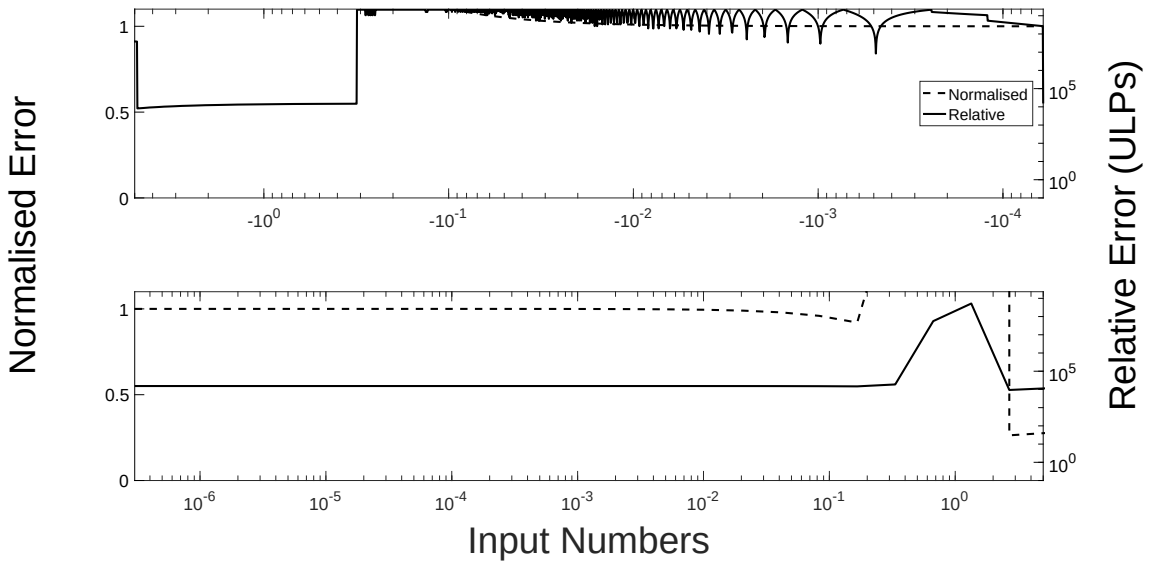


Figure 4.18: Power series approximation to 10 iterations in half-precision using a five stage Newton-Raphson division implementation. Top half of the graph shows error for a negative input, bottom half shows error for a positive input.

of resources, curve-fitting techniques were applied. Figure 4.21 shows the proposed flow for a curve-fitting approximation. The implementation converts the input floating-point number to fixed-point so that integer multiplication can be used. The result of the multiplication is de-constructed to obtain new exponent and mantissa values. Curve fitting is applied to adjust the new mantissa to more closely map to the exponential curve.

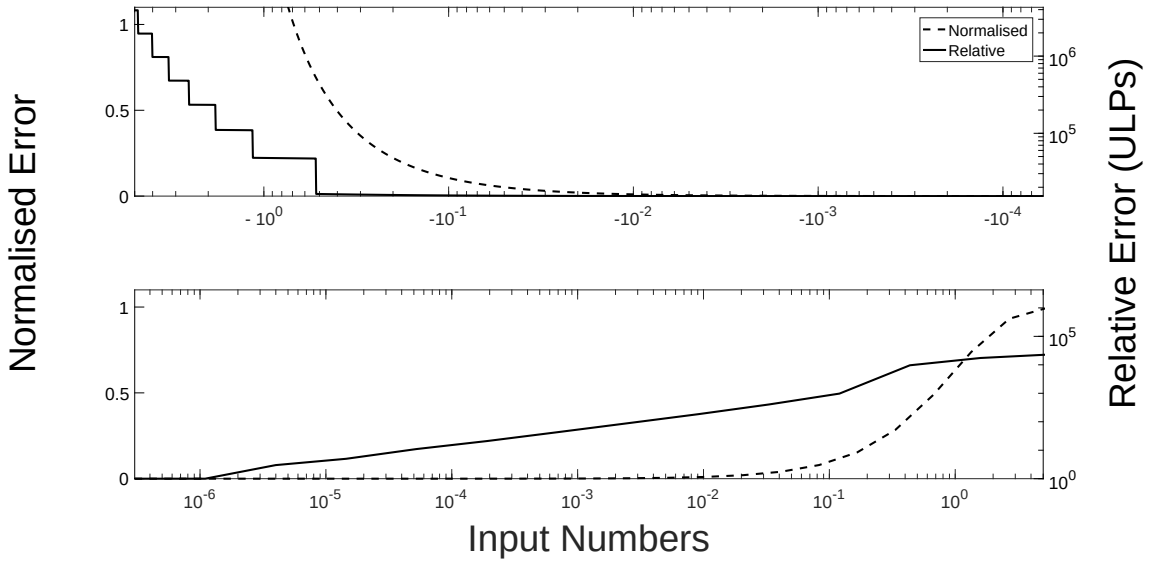


Figure 4.19: Euler series approximation to 10 iterations in half-precision using a single stage Newton-Raphson division implementation. Top half of the graph shows error for a negative input, bottom half shows error for a positive input.

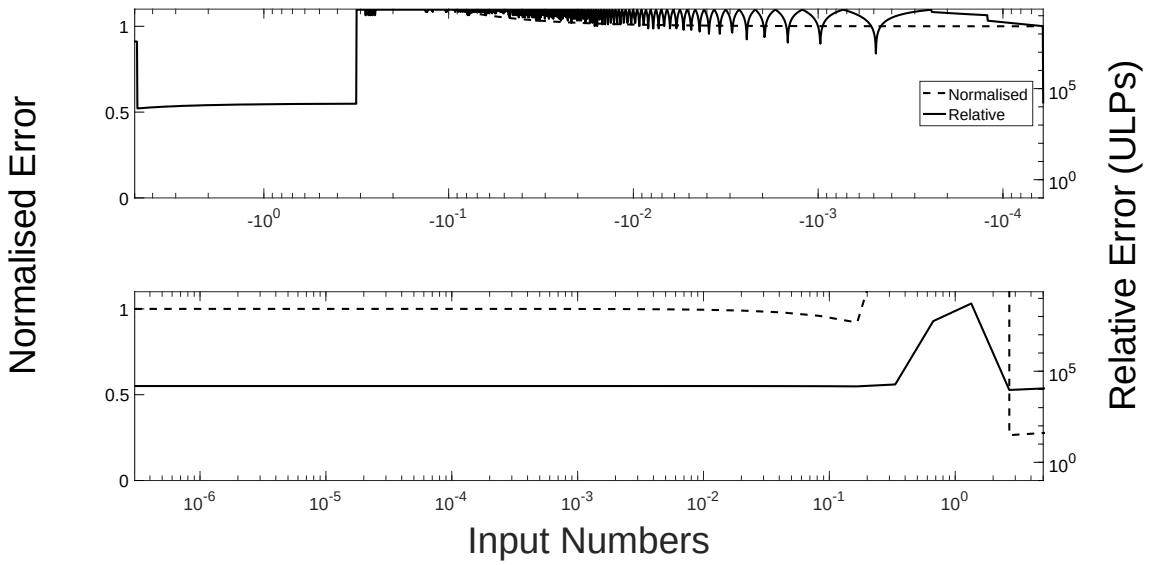


Figure 4.20: Power series approximation to 10 iterations in half-precision using a single stage Newton-Raphson division implementation. Top half of the graph shows error for a negative input, bottom half shows error for a positive input.

A number of different curve-fitting approaches have been used, detailed in Table 4.7. Linear piecewise approaches break the problem space into a number of sections (one, two and four) and use straight lines to map the output; quadratic and cubic approaches use a single, continuous function for curve mapping. In addition, a number of 'hybrid' approaches were also considered. These take the piecewise and continuous curve mapping approaches

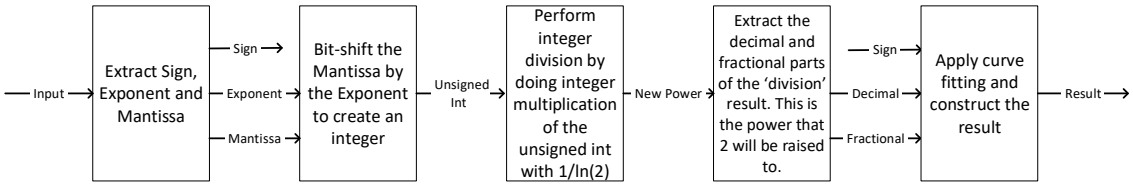


Figure 4.21: Flow diagram of the stages performed by the hardware implementation of the proposed hardware friendly exponential function.

and multiplex them with a small input approximation $(1+x)$. Based on the magnitude of the input the implementation switches between the two approximations. The result is therefore highly accurate for small input values, but has a larger resource cost. All implementations have also had pipelining stages added to increase f_{max}.

Table 4.8 shows the metrics for each implementation at half-precision to allow direct comparison to the Euler and power series approximations. The curve fitting methods have a very low resource cost compared to iterative methods. Some designs require fewer than 100 ALMs and registers. The maximum resource cost for any implementation is 107 ALMs and 169 registers at half-precision. (Appendix E shows metrics for single- and double-precision implementations.)

Figures 4.22 and 4.23 plot the number of registers and ALMs required in each of the 22 explored curve-fitting methods. Pipelining each design increases the required resources by a marginal amount, particularly for linear piecewise approaches. The largest resource penalty is incurred by the 'hybrid' designs (11-19), since they include a floating-point adder and control logic to select which approximation to use. Chapter 3 has already shown the costs for a floating-point adder (approximately 1000 ALMs, 500 ALMs and 250 ALMs for double-, single- and half-precision implementations respectively). However, the largest double-precision curve-fitting method presented still requires fewer resources than almost all half-precision Euler and power series implementations presented previously.

From Figures 4.22 and 4.23 it can be seen that there is a significant additional resource cost for quadratic and cubic curves, particularly when the design is pipelined. Increasing the order of the polynomial nominally adds one extra addition and three additional multiplications; when this is implemented in hardware it can be reduced to a single multiply-accumulate stage, shown in equation (4.24). From this it can be extrapolated that solving a fifth-order polynomial (as in standard C) would incur significant resource cost.

$$a + bx + cx^2 + dx^3 = x(x(dx + c) + d) + a \tag{4.24}$$

Table 4.7: Types of approximation for the exponential function in hardware.

Index	Description
1	Single piecewise linear approximation
2	Single piecewise linear approximation with pipelining stages
3	Double piecewise linear approximation
4	Double piecewise linear approximation with pipelining stages
5	Four piecewise linear approximation
6	Four piecewise linear approximation with pipelining stages
7	Quadratic approximation
8	Quadratic approximation with pipelining stages
9	Cubic approximation
10	Cubic approximation with pipelining stages
11	Hybrid approximation using $1+x$ and the single piecewise linear module depending on the magnitude of the input
12	Hybrid approximation using $1+x$ and the single piecewise linear module depending on the magnitude of the input with pipelining stages
13	Hybrid approximation using $1+x$ and the double piecewise linear module depending on the magnitude of the input
14	Hybrid approximation using $1+x$ and the double piecewise linear module depending on the magnitude of the input with pipelining stages
15	Hybrid approximation using $1+x$ and the four piecewise linear module depending on the magnitude of the input
16	Hybrid approximation using $1+x$ and the four piecewise linear module depending on the magnitude of the input with pipelining stages
17	Hybrid approximation using $1+x$ and the quadratic module depending on the magnitude of the input
18	Hybrid approximation using $1+x$ and the quadratic module depending on the magnitude of the input with pipelining stages
19	Hybrid approximation using $1+x$ and the cubic module depending on the magnitude of the input
20	Hybrid approximation using $1+x$ and the cubic module depending on the magnitude of the input with pipelining stages
21	Single piecewise linear approximation using a floating-point multiplier
22	2^x

Figure 4.24 plots the maximum operating frequency of each implementation. Each implementation has an issue rate of one, regardless of whether or not it is pipelined. Pipelining is particularly important when using polynomials for curve-fitting. Increasing the order of the polynomial without pipelining stages reduces f_{max}. This is due to having to perform a number of addition and multiplication operations in sequence inside a single clock period. Pipelining stages allow these operations to be spread over multiple periods, leading to

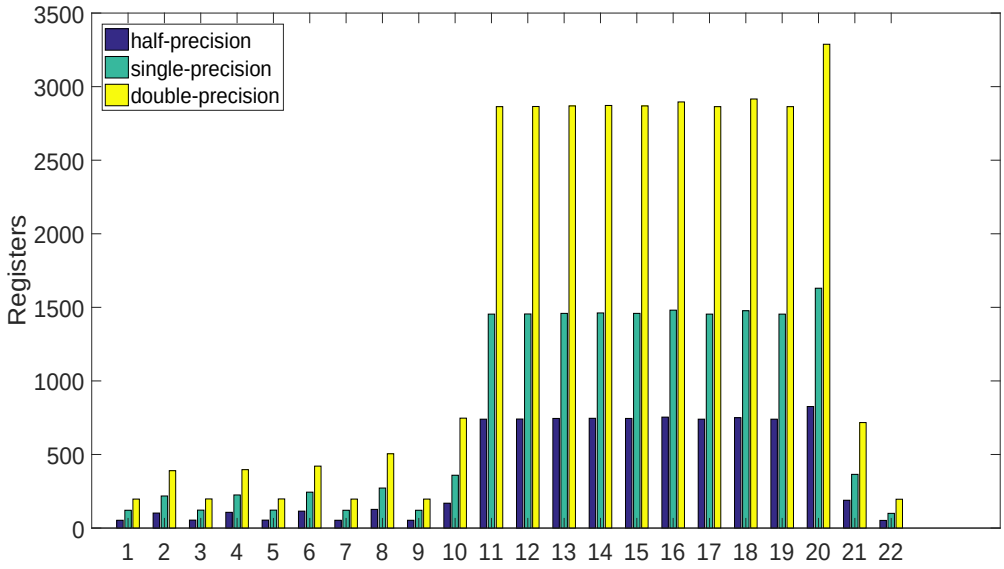


Figure 4.22: Registers required for hardware implementations of the exponential operation.

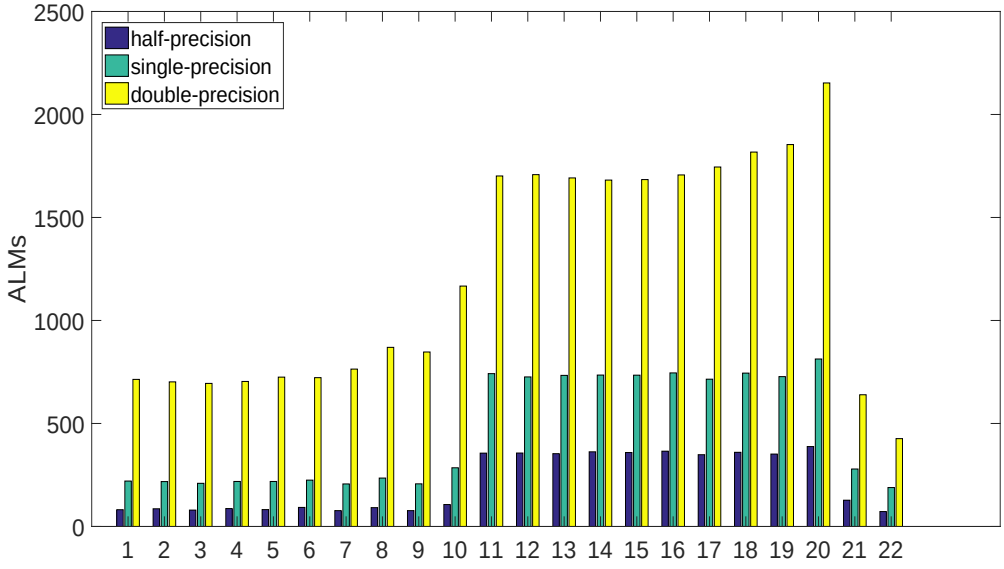


Figure 4.23: ALMs required for hardware implementations of the exponential operation.

a greater f_{max} at the expense of resource. Extrapolating to a fifth-order polynomial in a non-pipelined design results in a very low f_{max}, and consequently very low throughput. As has been seen with other functions presented in Chapters 3 and 4, Intel FPGA provide a floating-point exponential block that can be implemented on Cyclone V technology. Again this is a multi-cycle block that adds latency to a design: 17 cycles for single-precision or 25 cycles for double-precision. The resource count for the single-precision Intel exponential block is approximately 400 ALMs, and approximately 3,000 ALMs in double-precision. Both

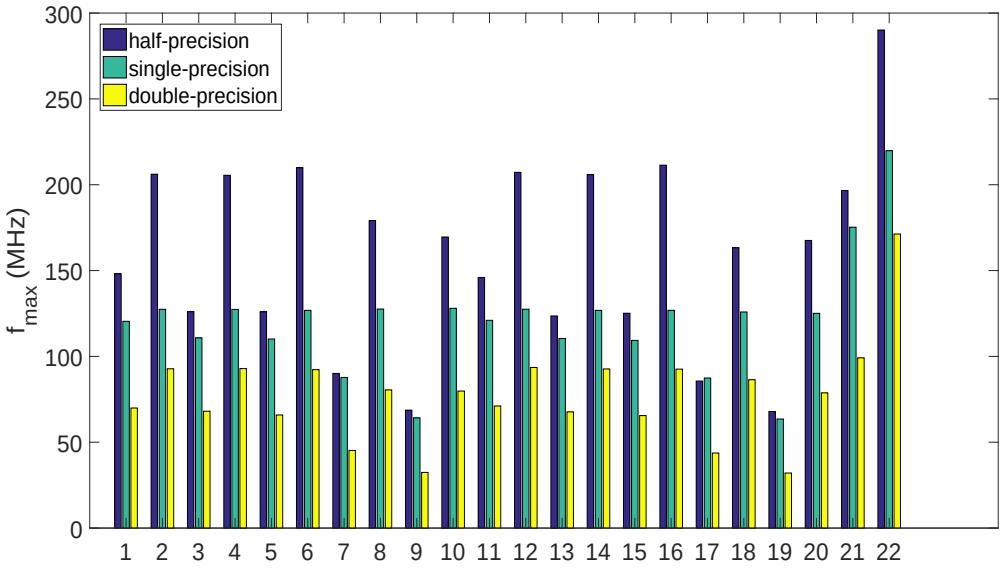


Figure 4.24: f_{max} of hardware implementations of the exponential operation.

single- and double-precision implementations require DSP blocks, with them using 19 and 46 respectively.

4.1.6 Analysis of error

The error plots, Figures 4.25 and 4.26, demonstrate that the implementations do not achieve one or fewer ULPs of error. The normalised error is always below one, often by several orders of magnitude. The description of the exponential function in *math.h*, states that the maximum error is bound to 2^{-59}. To achieve this accuracy a fifth-order polynomial is required. Depending on the application, the accuracy presented by these implementations may be sufficient. A case study presented later in this Chapter will demonstrate a working system using these approximations.

The implementation being used replaces e^x with 2^a, where $a = x/ln(2)$. a may be calculated using either a multiplication by $1/ln(2)$ (Figures 4.25 and 4.26) or a division by $ln(2)$. In the hardware, integer division has some limitations. The width of certain blocks on the device, such as integer divide, have a limit of 32-bits - this is the *lpm_width*. While it is possible to divide the integers created from half-precision floating-point inputs, the integers created from single- or double-precision are too wide for implementation in the hardware used. Figure 4.27 gives the simulated error for a double-precision system, should the integer division be realisable. The normalised error from the integer division implementation for negative numbers is orders of magnitude worse than if multiplication is used.

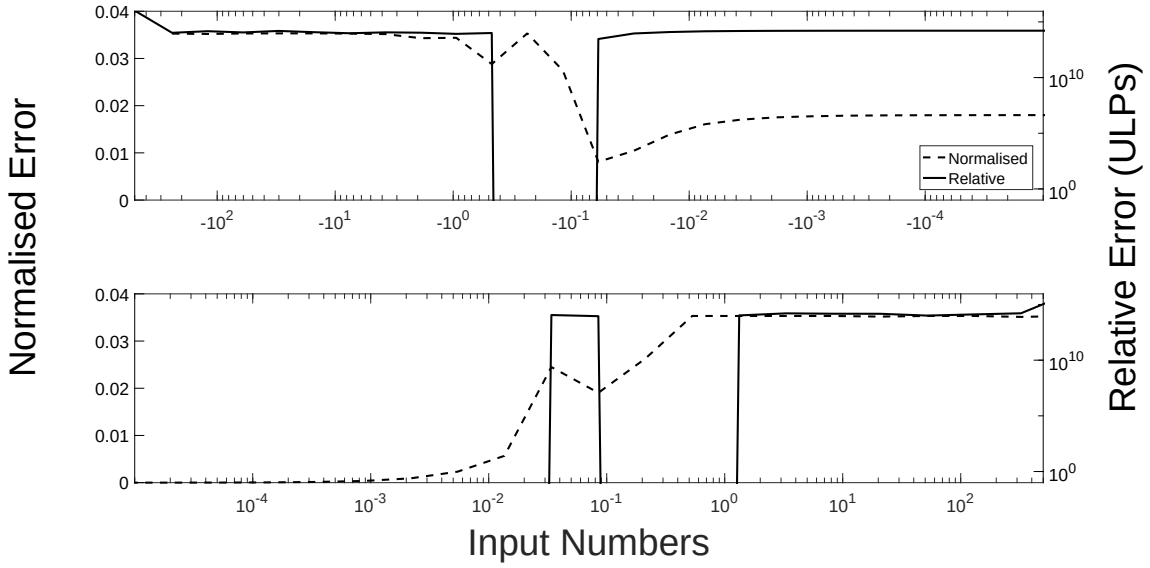


Figure 4.25: Hardware friendly floating-point exponent approximation using a single line curve fit in double-precision. The top graph is negative input numbers. The bottom graph is positive input numbers.

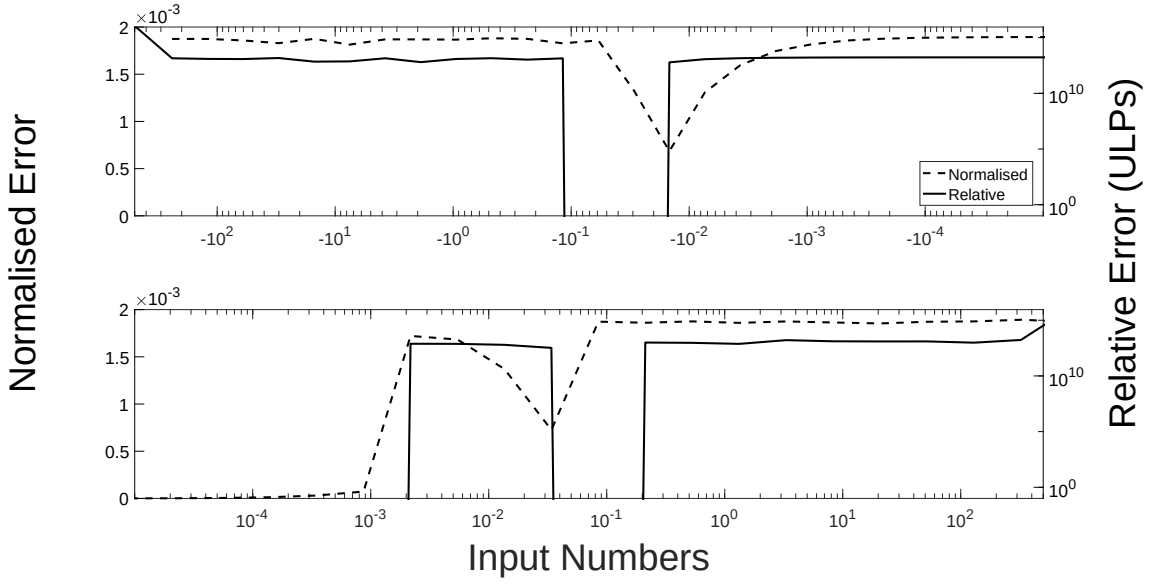


Figure 4.26: Hardware friendly floating-point exponent approximation using a four line curve fit in double-precision. The top graph is negative input numbers. The bottom graph is positive input numbers.

Chapter 3 presented designs for floating-point multipliers. Since the implementation shown here is for a floating-point exponential function, Figure 4.28 gives the error for an implementation that replaces the integer multiplication with a floating-point multiplication. Comparing this with the equivalent curve fitting implementation that uses integer multiplication, Figure 4.25, shows no difference in the error of the two techniques. However, the use of a floating-point multiplier incurs additional resource cost, shown in Figures 4.22 and 4.23. The integer multiplication method is considered better.

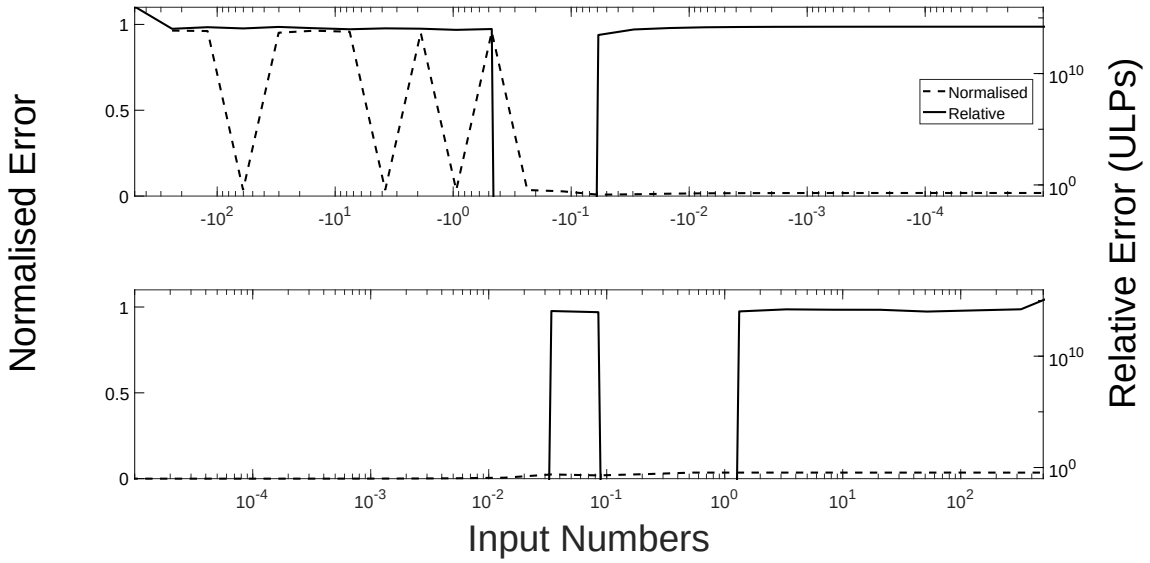


Figure 4.27: Hardware friendly floating-point exponent approximation using a single line curve fit with fixed-point integer division operation double-precision. The top graph is negative input numbers. The bottom graph is positive input numbers.

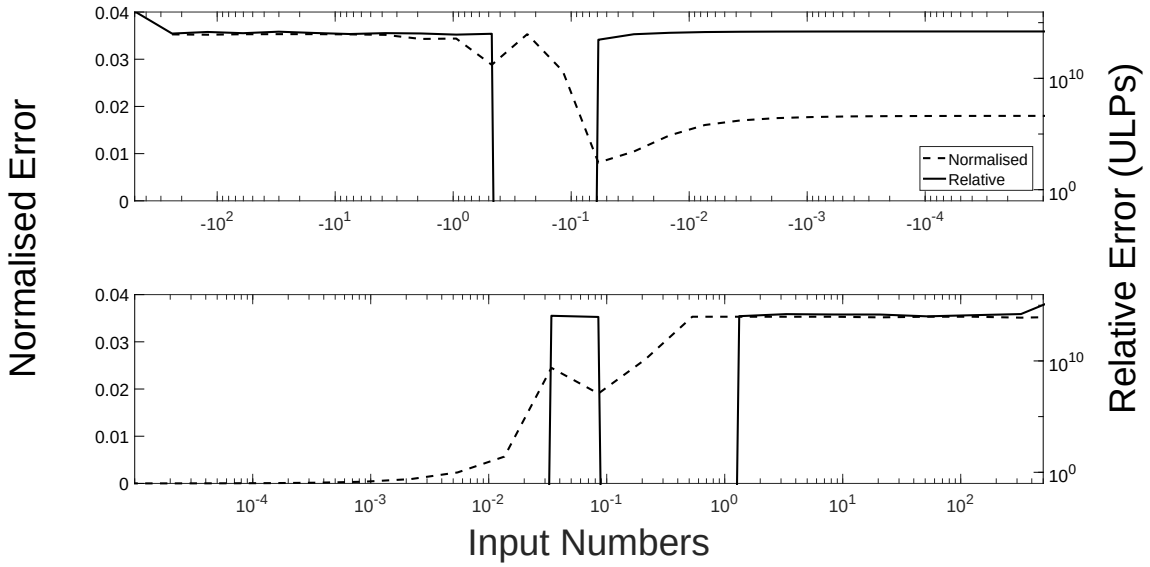


Figure 4.28: Hardware friendly floating-point exponent approximation using a single line curve fit with a floating-point multiply in double-precision. The top graph is negative input numbers. The bottom graph is positive input numbers.

Figures 4.29 and 4.30 show the error for two of the 'hybrid' exponential implementations. As expected the $1+x$ approximation for small inputs gives greater accuracy in the output over the small input region. The boundaries of the regions were calculated by finding the crossing points in error of each implementation and the $1 + x$ approximation. Greater accuracy for small inputs may be necessary for some applications, but this has a higher resource cost.

Hardware-friendly approximations of the *exponential* function are problematic. Methods for approximating the result are resource and time intensive. For systems that wish to

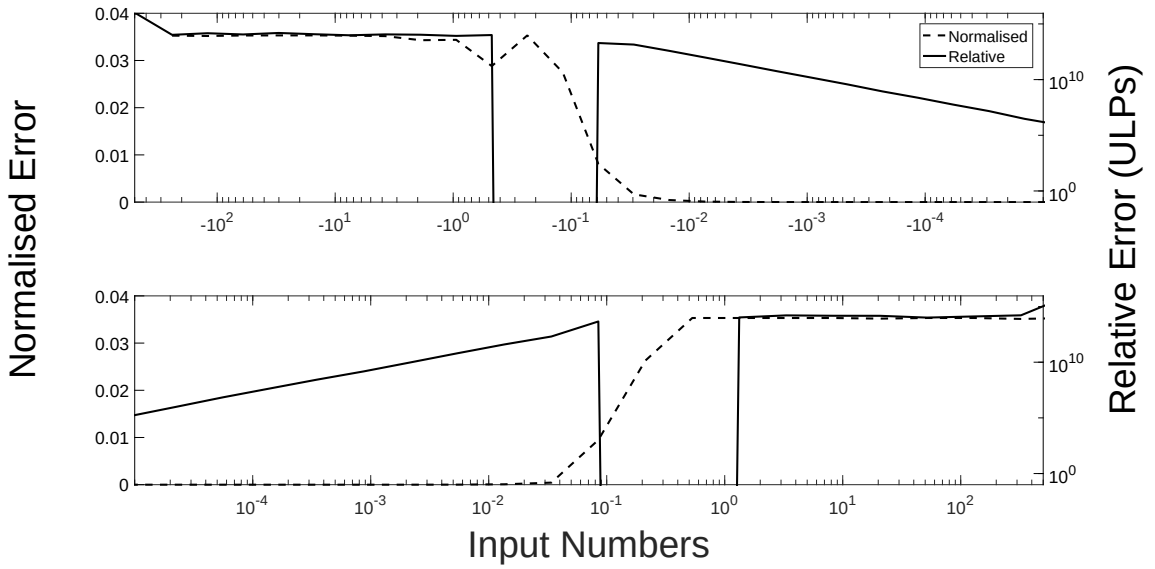


Figure 4.29: Hardware friendly floating-point exponent hybrid approximation single line curve fit and $1 + x$ in double-precision. The top graph is negative input numbers. The bottom graph is positive input numbers.

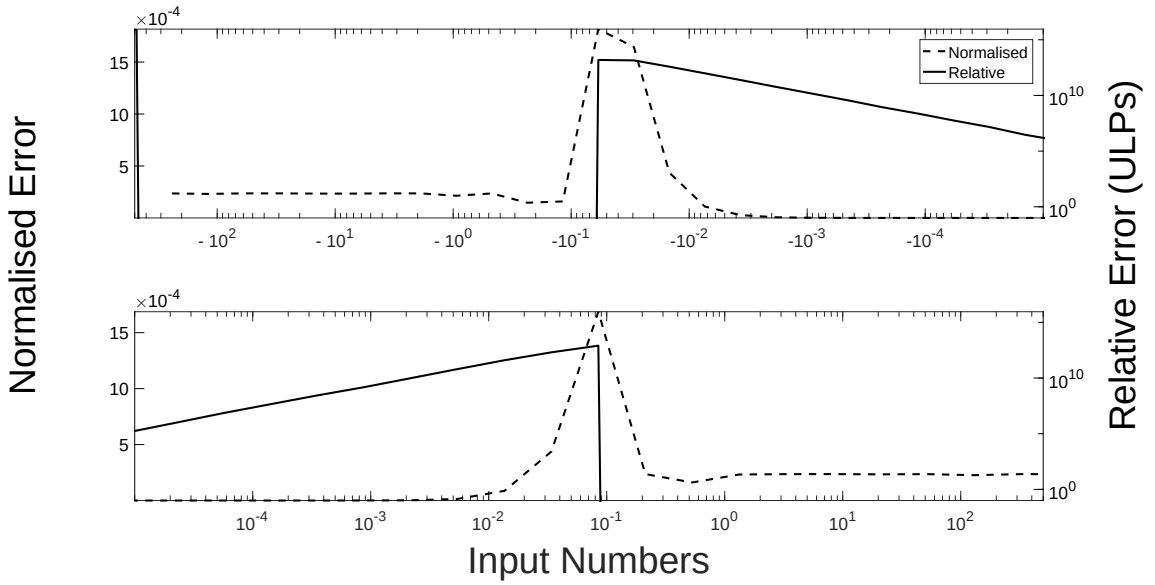


Figure 4.30: Hardware friendly floating-point exponent hybrid approximation cubic curve fit and $1 + x$ in double-precision. The top graph is negative input numbers. The bottom graph is positive input numbers.

benefit from hardware acceleration, this is impractical. The method used by a processor to increase the calculation speed and achieve one ULP or fewer relative error is still considered intensive for hardware implementation. Curve fitting methods that use piecewise or low-order polynomials were examined. Although the normalised error of these methods is low; the relative error can be large. A cubic polynomial curve-fit gives a relative error of four

or fewer ULPs for very small inputs (less than -10^{14}). For inputs greater than -10^{14}, the relative error is zero ULPS (shown in Appendix E). The trade-off is these systems can be run quickly, with an issue rate of one. The addition of pipelining resources adds overhead and allows a throughput of up to 200 Mega Floating-Point Operations Per second (MFLOPs). Depending on the application, saving resources - at the expense of accuracy - may be more important.

4.2 Case study: implementing a neuron in hardware

Hardware light methods for approximating $exp(x)$ use a re-arrangement of the function to the form 2^a. To test the validity of the approximation, a hardware implementation of the Hodgkin-Huxley neuron model was created [146]. The model uses several exponent functions. This case study will show the effect of the inaccuracy on a system that is governed by the shape of a non-linear function, rather than exact output values of the function.

Neural networks and the modelling of biological neurons are notoriously difficult. Biological neurons are analogue systems that do not map well to discrete digital architectures. A number of models for neurons exist (Hodgkin-Huxley [146] and Izhikevich [147]). This case study will focus on the Hodgkin-Huxley model. They are discrete approximations.

4.2.1 Training neural networks using approximations to the exponential function

For evaluation three identical neural networks that use different methods for the exponential in their transfer function were trained and evaluated against the MNIST database [148]. The three different networks use either a normal exponential, the single piecewise linear approximation or the quadratic curve approximation. Each network has an input layer, a hidden layer and an output layer. Each network was trained and tested 100 times. Figure 4.31 shows the scores of each network.

From Figure 4.31 it can be seen that using approximations for the exponential function produces a network that functions at least as well as a neuron that uses the traditional exponential function implementation, despite the large relative error. This confirms that the shape of the function is more important than the accuracy.

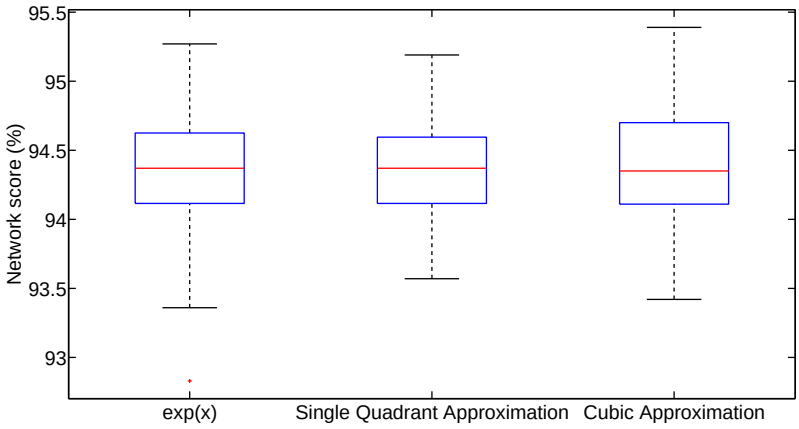


Figure 4.31: Performance of neutral networks that replace the exponential function with approximations of the exponential function that are more hardware friendly. Networks are trained and tested on the MNIST database. The network score is the number of times the network produced the expected answer while processing the test set after being trained.

4.2.2 Implementing the Hodgkin-Huxley model on an FPGA

Note: All results for double-precision floating-point implementations can be seen in Appendix F.

The Hodgkin-Huxley model of a neuron was implemented on an FPGA, using each of the exponential approximations. The implementations have been simulated to plot the output and membrane voltages. From the results it can be seen that the method for approximating the exponential function is application dependent.

Figures 4.32 to 4.35 show the resource and performance metrics for the implementations. The indices on the x-axis refer to the type of approximation being used for the *exponential* function as per Table 4.7.

The only variation between implementations is the exponent approximation; the resources being used for the rest of the model are constant. This is reflected by Figures 4.32 and 4.33. The resources required for the rest of the design dominates the resource variation introduced by changing the exponential approximation. Using Euler and power series methods for the exponent results in a design that is too large for the target Stratix V device. The curve fitting methods shown here require only approximately 40-50% of the ALMs available.

Even targeting a high-end Stratix device the operating frequency is very low, sub 0.35 MHz. However, the model is a discrete time model of a biological system, so the performance metric becomes 'simulated time per unit time'. At every time step the internal and resultant potentials of the neuron are calculated for a given stimulus. The timestep is a user input (a typical value is 0.1 s).

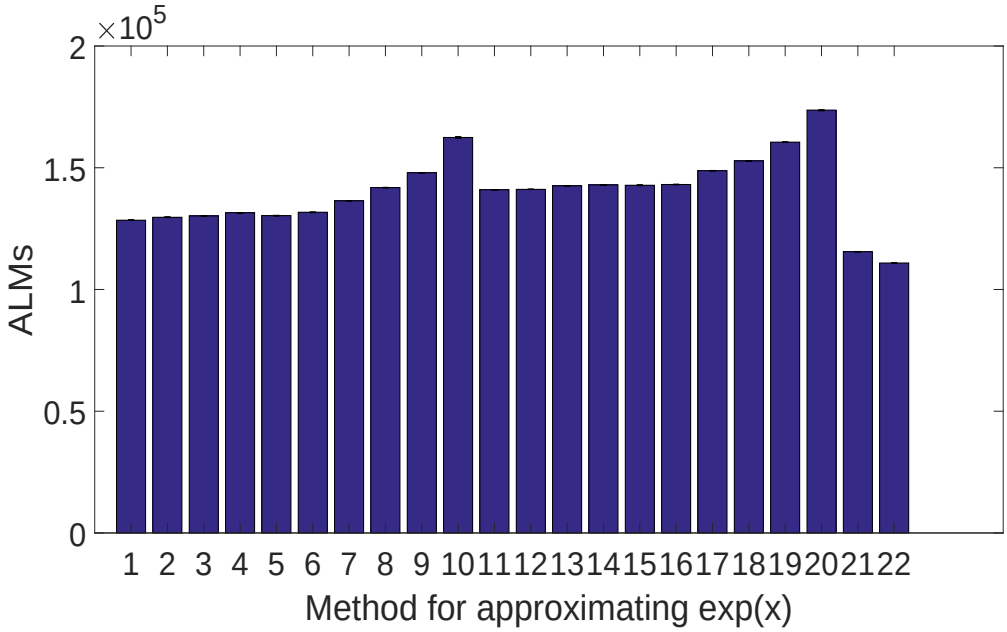


Figure 4.32: Using an array of different methods for implementing the exponential function on an FPGA, a study of the required number of ALMs for each model was performed. The models all work in double-precision floating-point.

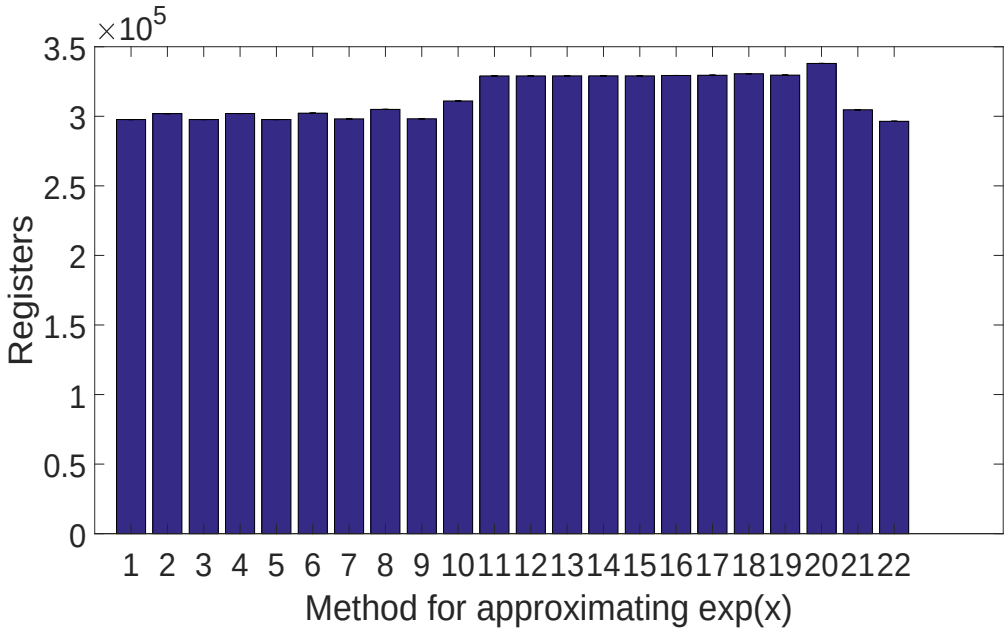


Figure 4.33: Using an array of different methods for implementing the exponential function on an FPGA, a study of the required number of registers for each model was performed. The models all work in double-precision floating-point.

Each implementation takes a fixed number of clock cycles to produce the next answer. The number of clock cycles for each design varies slightly due to the different latencies of the exponential approximations. For the different approximations this is between four and twelve clock cycles. Other factors that determine the latency of the design is the number

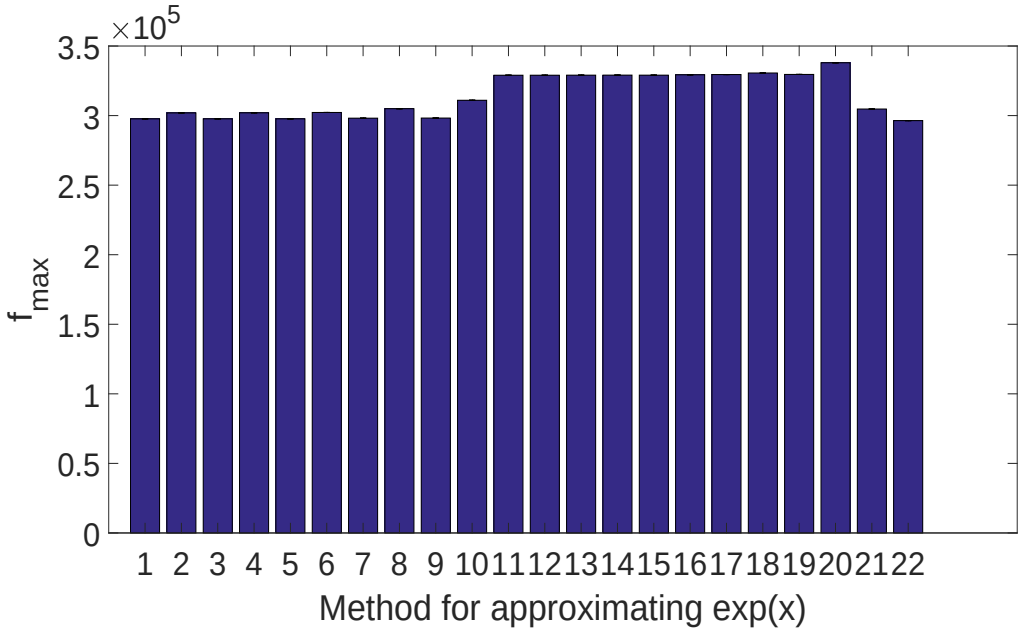


Figure 4.34: Using an array of different methods for implementing the exponential function on an FPGA, a study of the maximum operating frequency of each model was performed. The models all work in double-precision floating-point.

of any Newton-Raphson stages for reciprocation calculations (Number of stages × 80 clock cycles). Fixed delay through the system is 186 clock cycles. From this the total latency of the implementation is calculated. Using some typical values: Newton-Raphson stages equals *two*, f_{max} equals 320 kHz and exponential latency equals 8 the total system latency becomes:

$$186 + (2 \times 80) + 8 = 354 \,\text{clock cycles}$$

A new value is produced approximately every 0.001 seconds. With a typical time step of 0.1 s the performance of the system is $100\, s.s^{-1}$; 100 times faster than real time. This is equivalent to simulating 100 neurons in real time using time division multiplexing of the network, assuming it is deeply pipelined.

Osorio presented an approach to creating a hardware implementation of the Hodgkin-Huxley model in their 2018 paper [149]. Osorio's implementation used the 'Runge-Kutta' method to calculate the exponent, a cyclic method that iterates over itself until an answer is formed. Since the Runge-Kutta method can require many cycles, this is a very time-costly approach. Osorio says for the folded implementation, "32 cycles are needed for obtaining the initial values n, m, and h; 41 for the core of the iteration; and twelve cycles for equation (4.25)".

$$x' = x + (x_1 + 2 \times x_2 + 2 \times x_3 + x_4)/6 \tag{4.25}$$

x is replaced by the membrane voltages m, n and h.

Further the unfolded architecture requires "141 cycles" for the core. This leads to a large number of required clock cycles. Unfortunately the paper does not report the f_{max} of any of the designs, so it is unclear how this translates to real term functionality.

Osorio's implementation also required DSP blocks for both the folded and unfolded versions: 305 and 1196 respectively. From Figure 4.35 it can be seen that the maximum number of DSP blocks used in any implementation presented here is below 150.

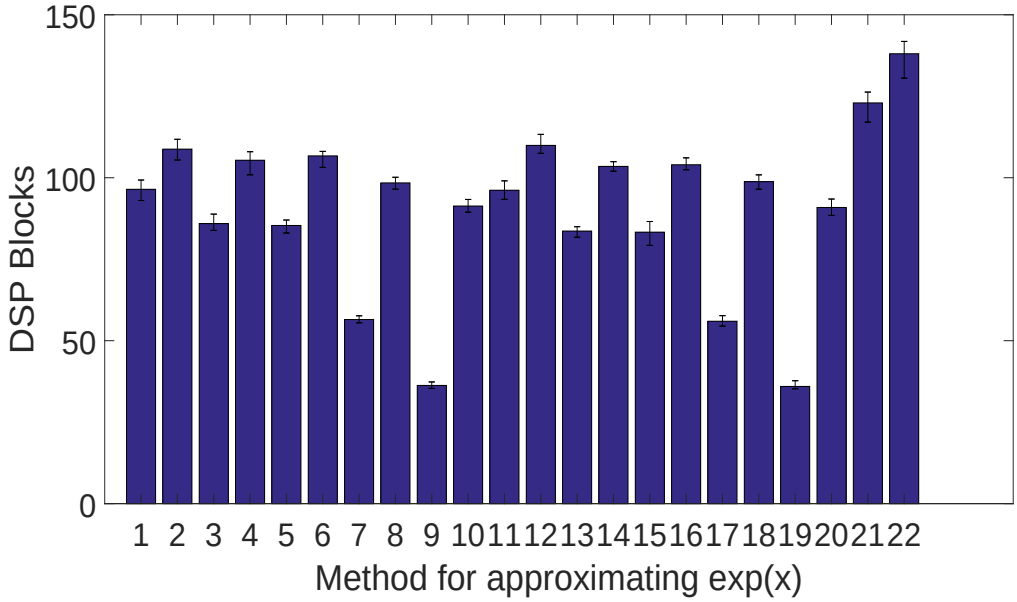


Figure 4.35: Using an array of different methods for implementing the exponential function on an FPGA, a study of the required number of DSP blocks for each model was performed. The models all work in double-precision floating-point.

The implementations of the Hodgkin-Huxley model presented in this Chapter have significant resource use and a very low f_{max}. Implementations 21 and 22 from Figure 4.35 have significantly higher DSP use compared to the other presented implementations. This is due to them using curve-fitting methods in the $exp(x)$ approximation. The high resource use for this model is unavoidable to an extent. As can be seen from equations (4.1 to 4.10), the Hodgkin-Huxley model comprises of a large number of operations, particularly *exponential* and *division* operations, which have high resource cost and lead to a low f_{max}. To mitigate the high resource count and low f_{max}, other methods for approximating $exp(x)$ could be used: for example, the implementations found in the Intel FPGA IP library that use a CORDIC algorithm. These implementations achieve an f_{max} of 284 MHz and 279 MHz, with 19 or 46 DSP blocks, for single- or double-precision respectively [126].

The FPGA implementation shown in this Section uses double-precision floating-point numbers. In practise a double-precision implementation is unlikely to be needed. In [149] Osorio discusses how the required precision for a mathematical function to remain correct can be analysed. This allows only the necessary level of precision to be used, hence saving resources. In Chapters 3 and 4 it is shown that lowering the precision yields a smaller implementation with higher f_{max} values. Therefore, a more practical implementation would use a mixture of precisions for each floating-point operation to provide the most optimal result in terms of throughput and area. The implementations for floating-point operations given in Chapters 3 and 4 are designed to operate with variable precision, set by a number of parameters during instantiation. Additionally, the design can be implemented with a deep pipeline to increase the f_{max}.

4.2.3 Outputs from the neuron simulation

Simulated outputs of the implementations are shown in Figures 4.36 to 4.38. The black line shows the stimulating current injected into the neuron. There are three internal potentials plotted for each neuron, labelled m, n and h, caused by the movement of sodium, potassium and calcium ions. The output voltage (v) of the neuron is plotted on the top graph. Figure 4.36 uses a single piecewise linear approximation which causes a mis-fire. However, the implementation does recover from this and continues to operate as expected. Figure 4.37 uses a piecewise linear approximation with two lines for curve fitting. There is now no mis-firing of the neuron and the output pulse train remains uniform. Increasing the accuracy of the approximation to the exponential function has no further impact on the functionality of the neuron. Figure 4.38 shows the impulse response of the same neuron as shown in Figure 4.37. The output matches the expected characteristic curve of the Hodgkin-Huxley model.

Neural networks represent an ideology in computing where a system can not only train itself to perform the desired task but has computational power equal to that of a human. The reality is that to achieve this level of power a vast number of processing resources are required. The SpiNNaker research project [150] is attempting to create specialist Application Specific Integrated-Circuit (ASIC) device that is modelled on the brain by networking together a number of ARM processor nodes. Although their devices are tailored for neural applications and are therefore better than processors at neural network applications, this technology still needs a significant amount of development.

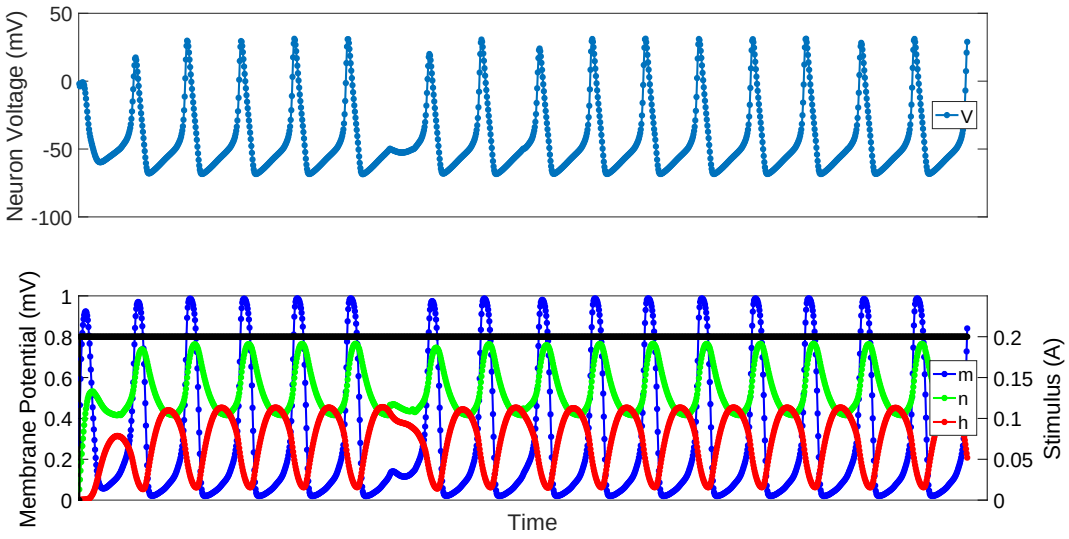


Figure 4.36: Hardware implementation of a Hodgkin-Huxley neuron using single line piecewise linear approximation for the exponential function. The neuron's response to a step input of 0.2 A (black trace, bottom graph). m, n, and h are the membrane potentials internal to the neuron.

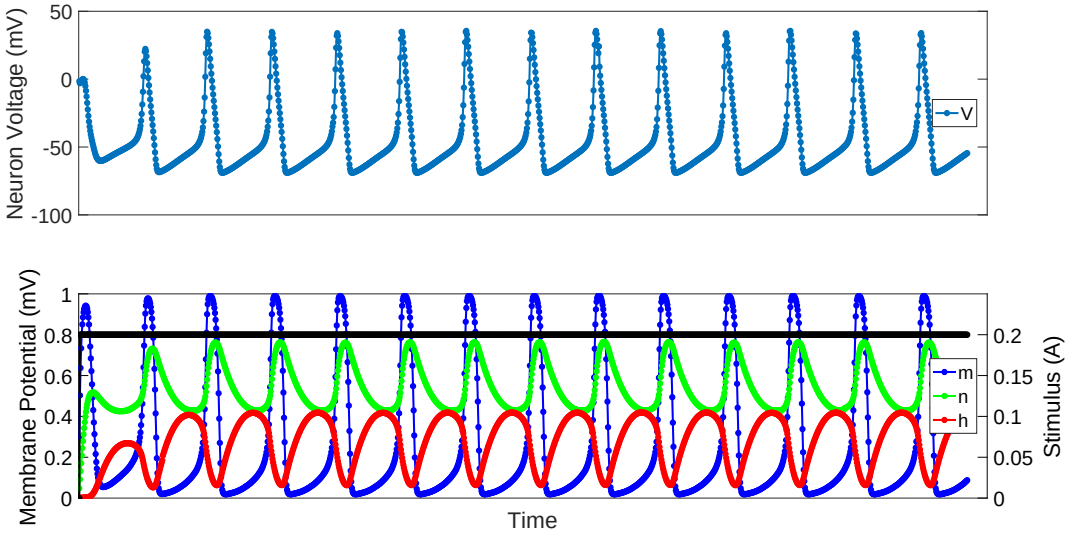


Figure 4.37: Hardware implementation of a Hodgkin-Huxley neuron using two line piecewise linear approximation for the exponential function. The neuron's response to a step input of 0.2 A (black trace, bottom graph). m, n, and h are the membrane potentials internal to the neuron.

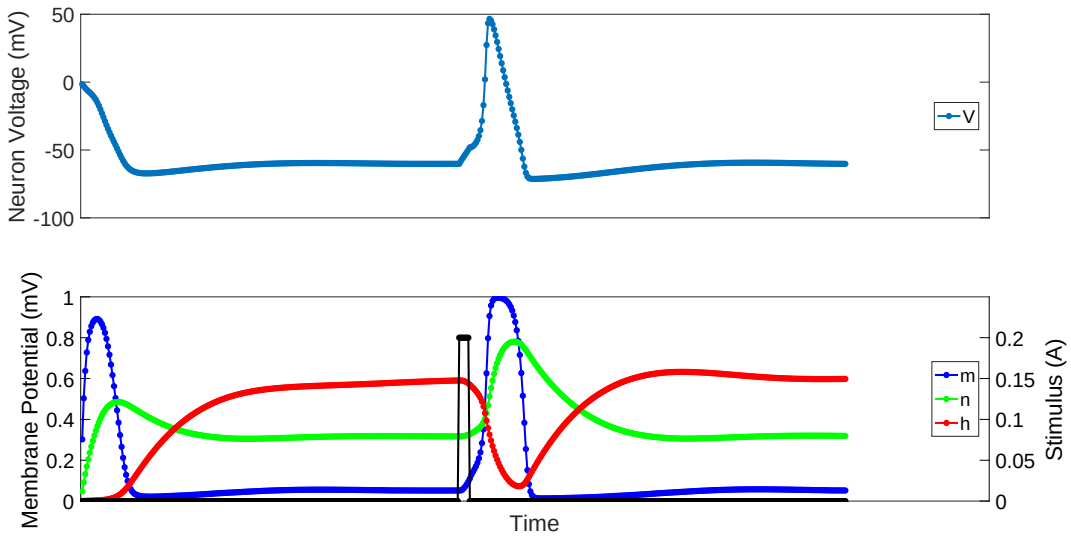


Figure 4.38: Hardware implementation of a Hodgkin-Huxley neuron using two line piecewise linear approximation for the exponential function. The neuron's response to a impulse of 0.2 A (black trace, bottom graph) lasting 50 μs. m, n, and h are the membrane potentials internal to the neuron.

4.3 Considerations for the implementations of other arbitrary complex functions

This Chapter has presented implementations for the reciprocal, square-root and exponential functions. It may be considered that if $x \in \mathbb{R}, \mathbb{C}$ in which case the magnitude of the Imaginary part of x is stored as a separate floating-point number. Computation for both the Real and Imaginary parts of the number may then be performed.

The Chapter discussed implementing a method for computing the exponential of a number. The inverse on the *exponential* function can also be approximated as in equation (4.26).

$$ln(x) = log\left(\frac{1+y}{1-y}\right) = 2\sum_{k=0}^{\infty}\frac{y^{2k+1}}{2k+1} \tag{4.26}$$

where $x = A \times 10^{n-1}$ and $y = \frac{A-1}{A+1}$. A is a number between one and ten, and n is the number of digits before the decimal point. Additionally, for small values $x \leq 1$, a Taylor-series expansion could be used equation (4.27).

$$ln(x) = (x-1) - \frac{1}{2}(x-1)^2 + ... \tag{4.27}$$

As before, hardware may be implemented to calculate the approximations.

4.4 Summary

Chapter 3 presented fundamental mathematical operations implemented in hardware. In this Chapter implementations for more complicated mathematical operations have been discussed. These functions were: reciprocal, divide, square-root and exponential. Three different methods of implementations have been examined. The reciprocal and divide operations were implemented using Newton-Raphon iteration. The square-root used non-restoring algorithms and the exponential used curve-fitting methods.

Successive approximation methods, such as the Newton-Raphson, used can lead to an answer that is within one ULP. The number of iterations required for convergence can be large. This results in either a high resource cost or low throughput. The Newton-Raphson method for double-precision floating-point converged within five iterations, however, the relative error was three ULPs.

Methods for approximating functions without iteration have been discussed. The square-root can be found using non-restoring algorithms. The non-restoring method splits the input number into pairs of bits and evaluates each pair in turn to extract the answer. Literature regarding non-restoring algorithms claims that the most accurate these algorithms can be is only half the number of bits of the input. This research has shown that if the mantissa is padded to double its length and the algorithm is adapted to reflect this, a much greater accuracy is achieved. Results from tests show a one ULP or smaller error in the result. It is now compliant with IEEE-754R. Additionally, the nature of non-restoring algorithms requires only very simple, logical functions that lend themselves well to hardware and achieve high throughput for low resource counts.

Implementing the exponential function is more challenging. Euler's number (e) does not lend itself to easy implementation in a binary system. It has been seen from *math.h* that e^x may be re-written in the form of 2^a, where a is scaled by the $ln(2)$, which is a constant. Processors work using base two, therefore 2^a is far easier to compute. Similar methods have been used for a hardware implementation. Separating the floating-point number into its exponent and mantissa allows the curve-fitting technique to be used over a small region, $1 \geq x < 2$. A number of curve-fitting techniques have been proposed and evaluated. These range from single linear piece-wise approximation to a cubic. Increasing the number of sections on a piece-wise approximation increases the accuracy at a trade-off in the number of resources. Curve-fitting using low order polynomials reduces the overall number of resources required compared to increasing the number of sections for a piece-wise approach. Most of

the approximations give a relative error more than one ULP, apart from the cubic curve-fit which gives a relative error of four or fewer ULPs. However, all implementations have a low resource count and high throughput. For a number of applications this error may not cause a problem. This has been demonstrated by the neuron case study.

Chapter 5 will use the hardware implementations from this Chapter and Chapter 3 to create a graphics processor on an FPGA. The FPGA-GPU is a heterogeneous system that uses the FPGA fabric for hardware acceleration. Trading-off accuracy for resource use for implementing a GPU on an FPGA will be discussed.

Table 4.8: Resource requirements and timing analysis for hardware efficient implementations of approximations of e^x. Implementations are using half precision floating point accuracy.

Module	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs Recoverable by Dense Packing	[C] Estimate of ALMs Unavailable	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
Single quadrant	81.5 (81.5)	85.5 (85.5)	4.0 (4.0)	0.0 (0.0)	136 (136)	53 (53)	2	148.08	147.6
Single quadrant with pipeline	86.0 (86.0)	111.5 (111.5)	25.5 (25.5)	0.0 (0.0)	144 (144)	102 (102)	2	206.1	199.2
Double quadrant	79.5 (79.5)	82.0 (82.0)	3.0 (3.0)	0.5 (0.5)	136 (136)	54 (54)	2	126.06	126.49
Double quadrant with pipeline	87.0 (87.0)	108.0 (108.0)	21.0 (21.0)	0.0 (0.0)	149 (149)	107 (107)	2	205.51	197.82
Four quadrant	82.0 (82.0)	83.5 (83.5)	1.5 (1.5)	0.0 (0.0)	141 (141)	54 (54)	2	125.96	125.39
Four quadrant with pipeline	92.5 (92.5)	116.0 (116.0)	23.5 (23.5)	0.0 (0.0)	161 (161)	115 (115)	2	209.91	202.51
Quadratic fit	77.0 (77.0)	86.5 (86.5)	9.5 (9.5)	0.0 (0.0)	129 (129)	53 (53)	4	90.07	88.44
Quadratic fit with pipeline	91.5 (91.5)	126.0 (126.0)	34.5 (34.5)	0.0 (0.0)	147 (147)	127 (127)	4	179.08	181.39
Cubic fit	77.0 (77.0)	84.0 (84.0)	7.0 (7.0)	0.0 (0.0)	129 (129)	53 (53)	6	68.68	67.62
Cubic fit with pipeline	106.5 (106.5)	153.0 (153.0)	46.5 (46.5)	0.0 (0.0)	180 (180)	169 (169)	6	169.55	172.18
Hybrid using single quadrant	356.0 (39.0)	471.0 (43.3)	115.0 (4.3)	0.0 (0.0)	450 (49)	740 (35)	2	145.99	146.41
Hybrid using single quadrant with pipeline	356.5 (41.0)	478.5 (53.2)	123.5 (12.2)	1.5 (0.0)	454 (51)	741 (51)	2	207.21	201.25
Hybrid using double quadrant	353.5 (38.8)	477.5 (44.2)	124.0 (5.3)	0.0 (0.0)	451 (48)	745 (35)	2	123.56	123.61
Hybrid using double quadrant with pipeline	362.5 (41.7)	494.0 (52.1)	132.0 (10.4)	0.5 (0.0)	462 (51)	746 (51)	2	205.93	202.63
Hybrid using four quadrant	359.0 (39.4)	481.0 (44.7)	122.0 (5.3)	0.0 (0.0)	458 (50)	745 (35)	2	125.13	125.49
Hybrid using four quadrant with pipeline	365.5 (41.8)	513.0 (53.3)	147.5 (11.6)	0.0 (0.0)	475 (51)	754 (51)	2	211.37	205.72
Hybrid using quadratic fit	348.5 (38.0)	473.0 (45.2)	124.5 (7.2)	0.0 (0.0)	443 (49)	740 (35)	4	85.62	84.21
Hybrid using quadratic fit with pipeline	360.0 (39.3)	495.5 (45.9)	139.0 (7.9)	3.5 (1.3)	454 (49)	750 (35)	4	163.32	165.84
Hybrid using cubic fit	351.5 (36.8)	468.5 (41.8)	118.0 (5.0)	1.0 (0.0)	443 (49)	740 (35)	6	67.87	66.76
Hybrid using cubic fit with pipeline	388.0 (39.0)	526.5 (43.2)	138.5 (4.2)	0.0 (0.0)	495 (49)	826 (35)	6	167.53	166.75
Single quadrant with floating point multiply	127.5 (74.2)	163.0 (79.0)	35.5 (4.8)	0.0 (0.0)	225 (120)	189 (52)	1	196.58	196.0
Two to the power x	72.5 (72.5)	81.0 (81.0)	8.5 (8.5)	0.0 (0.0)	122 (122)	52 (52)	0	290.11	287.85
single quadrant using integer divide	489.0 (87.3)	493.5 (91.5)	4.5 (4.2)	0.0 (0.0)	948 (128)	81 (81)	0	24.78	24.27

Chapter 5

Case Study: Creating an OpenGL Compliant GPU on an FPGA-SoC

In Chapters 3 and 4, a number of hardware implementations of mathematical functions were presented. In isolation, these functions can provide floating-point acceleration for a processor. While there are many applications that can make use of reconfigurable logic, this Chapter will explore graphics processing. Graphics processing is accepted to be computationally intensive. The architecture of a Graphics Processing Unit (GPU) differs from a General Purpose Processor (GPP) in that it is optimised for processing large amounts of data in parallel. Moreover, due to the GPU's high data processing capability more tasks are being implemented on the architecture, seeking better performance. However, as they are not native graphics tasks, there is a loss in performance compared to native GPU tasks.

This Chapter will demonstrate the construction of an OpenGL compliant GPU using the reconfigurable fabric of the Field Programmable Gate Array (FPGA). Discussions will be presented regarding the benefits and limitations of using the reconfigurable fabric.

5.1 Replacing processors with dedicated hardware

Processors and dedicated hardware are different platforms. Processors are formed from a set of 'blocks'. At the basic level these blocks include an Arithmetic Logic Unit (ALU), program counter, registers and Input/Output (I/O). Creating these blocks provides a level of abstraction which reduces the efficiency of the device - for instance a processor requires memory operations. Hardware can increase the efficiency by implementing dedicated architectures, tailored to solving a specific problem, at the cost of flexibility.

It is possible to create a processor using reconfigurable hardware: [151, 152, 153] are a few examples. Dedicated hardware functions very quickly and efficiently but at the cost of flexibility.

5.1.1 Overview of a GPU

A GPU uses the Single Instruction Multiple Data (SIMD) topology, allowing the same operation to be applied to an array of data in parallel, increasing the throughput. Operations for GPUs are constructed as 'shaders' that are loaded to the device at runtime.

The general construction of an Open Graphics Language (OpenGL) based graphics pipeline can be seen in Figure 5.1a. The graphics pipeline consists of three basic stages: vertex processing, geometry processing, and pixel processing [154].

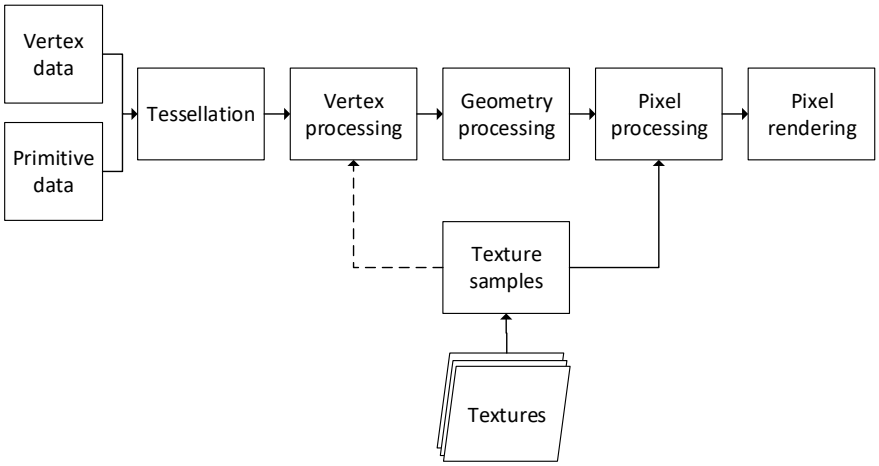
Vertex Processing (vertex shader): this section is responsible for co-ordinate space transforms. These include the world transform for positioning and rotating objects in world space; the view transform to move the vertices in the view space; and the projection transform to convert the 3D triangles and polygons into a 2D image. Additionally, the vertex shader can be responsible for animation techniques and light and colour computations.

Geometry Processing: the geometry processing stage converts vertex data into pixel data using rasterization - scanning through each pixel location and performing calculations to determine if it lies inside the triangle. It can also be used to perform clipping and culling operations that remove objects or parts of objects that are off screen or obscured by something else.

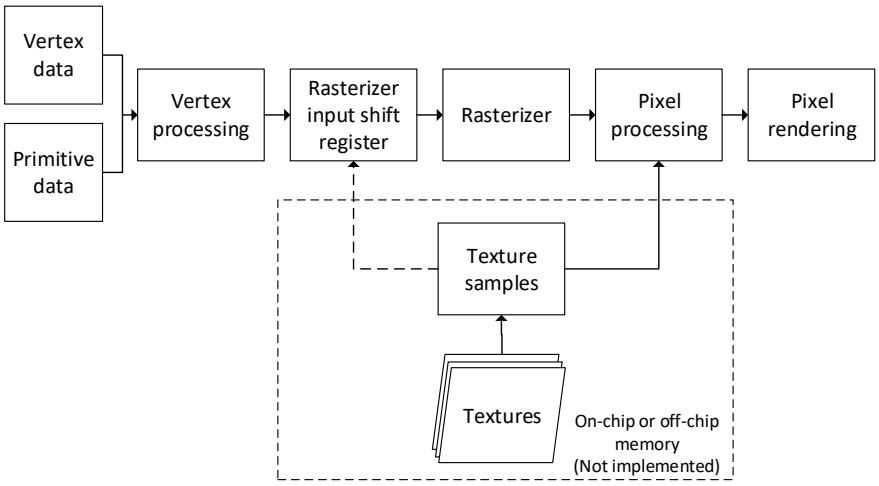
Pixel Processing (fragment shader): this is used to compute the colour of a single pixel. Colour computations consider a variety of different sources such as textures, ambient light, directional light, shadows and materials.

The functionality of these blocks has been replicated using hardware, shown in Figure 5.1b. There are a number of differences between the OpenGL pipeline and the FPGA based pipeline. The FPGA implementation includes a rasterization block (part of the geometry processing section). A separate module is used to control the type of rendering being used, for example *GL_TRIANGLES*, *GL_TRIANGLE_STRIP* or *GL_TRIANGLE_FAN*.

Graphics rendering processes often apply textures to a surface that has been drawn. Textures are passed to the graphics processor as an image file. For the FPGA implementation, these would be stored in either on- or off-chip memory (currently not implemented).



(a) OpenGL pipeline



(b) FPGA pipeline

Figure 5.1: The OpenGL pipeline can be broken down into a simple flow consisting of front end vertex processing, geometric processing (including rasterizing) and fragment processing, 5.1a. Primitive, or vertex, data describe the location of the triangles using co-ordinates which are generated by the host processor. To replicate this behaviour and create a compatible FPGA based implementation a similar system flow was used, 5.1b.

5.2 Implementing the FPGA-GPU

Before designing for the FPGA, a basic OpenGL render engine was constructed. Additionally, implementations of individual routines were modelled using C. The design was simulated using Model-Sim and implemented on an Intel Cyclone V FPGA [155].

5.2.1 Basic render engine

The basic render engine was constructed based on the guides found in [156]. The render engine consists of several managers to control the scene, models, and shader. The managers

were used to create a variety of simple objects that were transformed in the view space, coloured and textured.

The render engine was used to investigate the effects of different methods of shader construction. The engine was ported between different platforms and architectures, allowing for measurements of performance and power consumption of a variety of embedded platforms performing graphics rendering. This data was used to compare the performance of the FPGA implementation of a GPU later in this Chapter.

An example of the render engine's output can be seen in Figure 5.2.

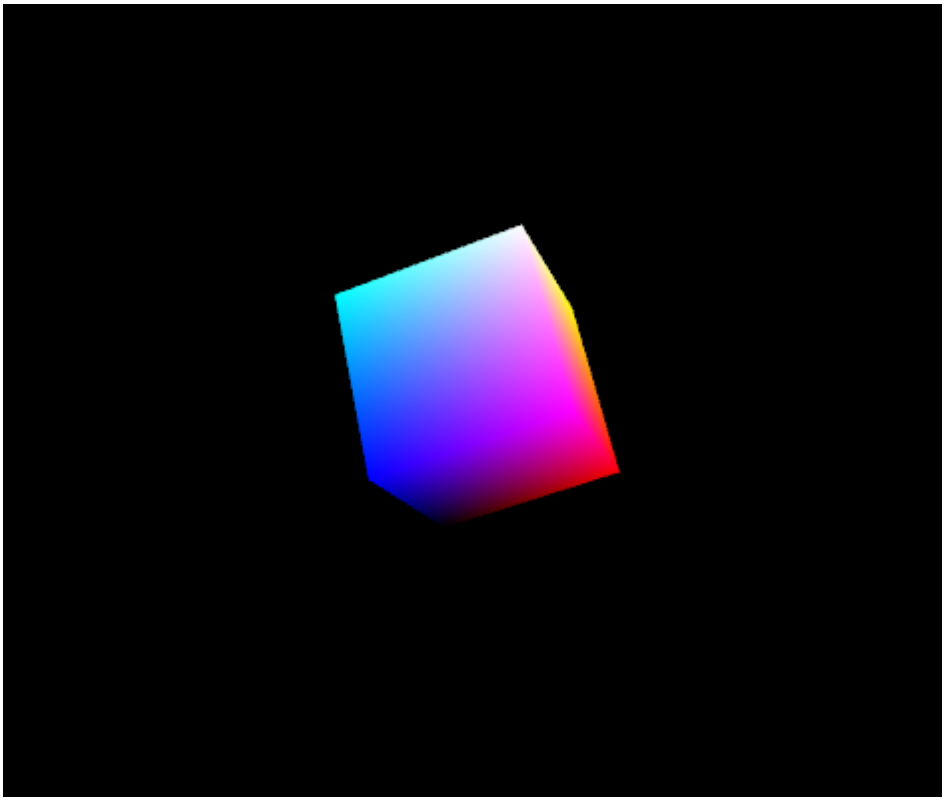


Figure 5.2: The OpenGL engine rendering a rotating cube. The faces are gradient shaded based on the Red-Green-Blue (RGB) values of the vertices.

5.2.2 Modelling the system

The graphics pipeline includes a process that converts primitive or vertex data into an array of pixels. The pixels are members of the object described by the vertices. This process is performed by the rasterization function. Before implementation in Hardware Description Language (HDL), it was first modelled in a procedural language.

Rasterization can be achieved using a number of different algorithms. The process used in the FPGA implementation is based on the Barycentric co-ordinate system [157]. The Barycentric co-ordinate system determines if a pixel lies inside a triangle and provides data

for interpolating the exact location inside the triangle. Data from the interpolation is used in the fragment shader when applying surface effects that are positionally dependent, such as light or texture maps.

5.2.3 FPGA implementation of the GPU

The complete FPGA implementation for the pipeline can be broken down into the following modules:

- Communication between processor and FPGA
- Vertex shader
- Vertex assembler (rasterizer feed)
- Rasterizer
- Fragment shader
- Pixel buffer and display controller

Everything between the vertex shader and the fragment shader constitutes the hardware equivalent implementation of the GPU. The front- and back-end processes are used to stream the data between different domains. The design was implemented on a FPGA-SoC device that contains a Hard Processor System (HPS), in this case an ARM Cortex-A9 processor. The processor streamed vertex values into the FPGA. The devices are on different clock domains, so clock domain crossing techniques were used to move the data safely without loss. At the back end of the GPU, data was fed into a soft processor to write to external memory. A pre-designed Video Graphics Array (VGA) controller and dual clock First-In, First-Out (FIFO) from the Altera IP library were used. The decision to use pre-existing IP was made to save time as memory interfaces are already known and it was not considered to be part of the FPGA-GPU design. There were further restrictions in the external memory and video interface due to the availability of development equipment.

5.2.3.1 Rasterizing unit

The rasterization unit converts the vertices, after they have been transformed by the vertex shader, into a stream of pixel values for the fragment shader. The fragment shader then applies lighting, colour and texture to each pixel.

Code listing 5.1 describes the algorithm for the rasterizer. Each mathematical operation has already been implemented in Chapters 3 and 4. The rasterization module receives sets of three vertices, each of which is in floating-point format. The output pixel values are integers.

```
1.  Always  find max X & Y and min X & Y,
2.  Always  find corner vertex,
3.  Always  select raster scan direction,
4.  Always  calculate edge vectors (vs1 & vs2),
5.  Raster scan over triangle bounding box
6.  Always  calculate Q vector
                (i − p(0), j − p(1)),
7.  Always  calculate cross products,
8.  Always  calculate Barycentric co-ordinate pair,
9.  if(S ≥ 0, T ≥ 0, S+T ≤ 1)
            Pixel inside triangle,
    else    Pixel outside triangle,
10. Repeat steps 6 - 9 over entire triangle
```

Listing 5.1: Algorithm for the operation of the rasterization unit to convert a set of three vertices to a stream of pixel value: Q is a vector; i, j and p are pixel locations; S and T are two of the three Barycentric co-ordinate values.

A traditional Barycentric rasterizer determines a bounding box for the input triangle, reducing the pixels values to be scanned through. Each position inside the bounding box are used in equations (5.1) to (5.3) to determine if the pixel is a member of the triangle.

The Barycentric method calculates two edge vectors, vs_1 and vs_2. These vectors remain constant over the entire triangle. A third vector, Q, represents a vector made from the current pixel being analysed and one of the input vertices, equation (5.1). The Barycentric method does not rely on the vertices being in a certain order. This reduces the complexity of implementation, using fewer resources.

$$\begin{aligned} vs_1 &= (v_2(x) - v_1(x), v_2(y) - v_1(y)) \\ vs_2 &= (v_3(x) - v_1(x), v_3(y) - v_1(y)) \\ Q &= (i - v_1(x), j - v_1(y)) \end{aligned} \tag{5.1}$$

v_x is a vertex of the input set and i, j is the location of the current pixel. Cross product values (CP_x) of the vectors (vs_1, vs_2, and Q) are calculated, as per equation (5.2). The cross

product of two 2×1 vectors $(x \times y)$ is given by $CP = x(1).y(2) - x(2).y(1)$.

$$\begin{aligned} CP_1 &= Q \times vs_2 \\ CP_2 &= vs_1 \times Q \\ CP_3 &= vs_1 \times vs_2 \end{aligned} \tag{5.2}$$

The ratio of the cross products, equation (5.3), are used to form two members of the Barycentric co-ordinate. If these fulfil the conditions that $S \geq 0$, $T \geq 0$, and $S + T \leq 1$ the pixel is known to be inside the triangle.

$$\begin{aligned} S &= CP_1/CP_3 \\ T &= CP_2/CP_3 \end{aligned} \tag{5.3}$$

All the mathematical functions required by the rasterizing algorithm have been covered by the functions given in Chapters 3 and 4, including the comparison and typecast operations.

5.2.3.2 Optimisations for the rasterizer

The rasterization operation is a computationally intensive process. There are a significant number of calculations with a potentially large search space. The throughput of this module can create a system bottleneck for the entire implementation. The system cannot move on to rendering the next triangle before completing a raster scan of the current triangle. Consequently, it was important that the implementation was optimised.

There are a number of optimisations that can be applied: drawing the bounding box, identifying calculations that can be performed once, and parallelisation.

The bounding box reduces the number of pixels subject to the raster scan, hence saving computation time. This can be taken further: placing a bounding box around a triangle will result in at least one vertex of the triangle existing in the corner of the box. This is set as the start position of the raster scan and the scan directions are updated to reflect this. The gradient of the triangle's edges are then determined by noting the i and j values when crossing into and out of the triangle. This information is fed back to update the start locations for new rows, shown in Figure 5.3. This eliminates processing pixels that are known to be outside of the triangle, reducing computation and increasing throughput.

It has also been identified that certain calculations (location of the bounding box, start position of the scan and the edge vectors vs_1 and vs_2) need to only be performed once per triangle. Other calculations such as the cross products (CP_x) and the Barycentric pair (S

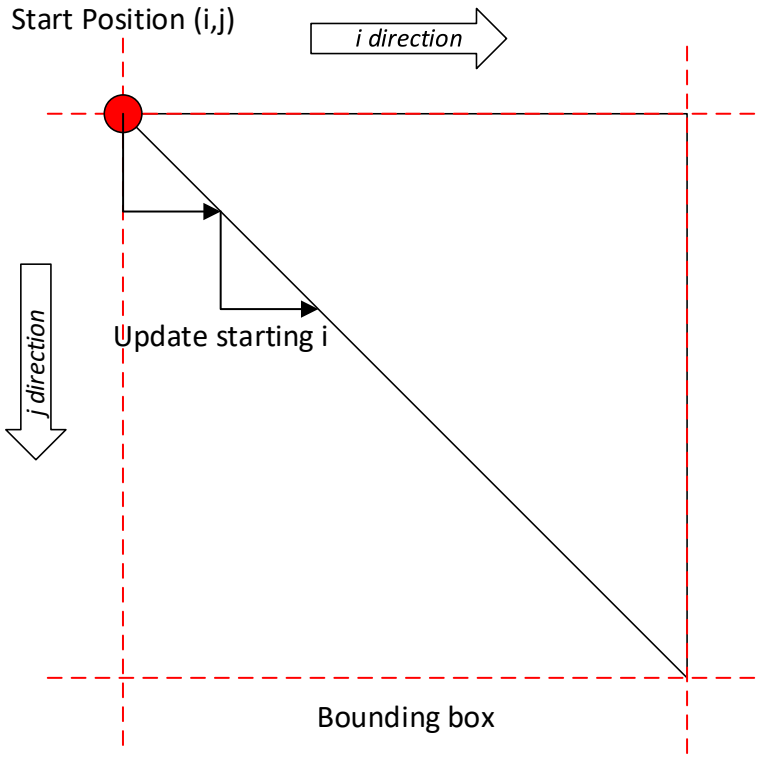


Figure 5.3: The heart of a graphics processor, a rasterizer is used to translate vertex data into the fragment (triangle) that is represented. Optimising this process reduces the number of redundant calculations performed by the system and increases throughput. This design uses gradients to reset the start or end points of the raster scan on a line by line basis.

and T) can be performed in parallel with each other.

Even with the optimisations and designing for an issue rate of one, the rasterization process creates a bottleneck. To further reduce bottlenecking, multiple rasterization modules can be instantiated. Multiplexing allows the system to select which rasterizer to feed a vertex set into.

5.2.3.3 Shaders and a common interfacing system

Graphics processors use 'shaders'. Shaders define the transformations and surface effects to be applied to an object being rendered. Shaders are compiled and loaded to the graphics processor at runtime. The FPGA implementation uses a standard interface to which all shaders conform. The standard interface allows a variety of shaders to be loaded without changing other parts of the design.

The port list for the shader implementation is broken into two sections: one controls the hardware flow and one controls the data flow. The hardware control ports are consistent across all the modules. The control signals are: *clock*, *reset*, *enable*, *valid*, *busy* and *wait*.

All modules have a *synchronous* reset. When a module is presented with the *enable* signal, data on the ports is considered valid and to be operated on. Once the module has completed calculations, it drives a *valid* signal high to indicate that the data on the output ports is ready. The valid signal is fed into the next module's *enable* port.

The *busy* and *wait* ports allow modules that reuse logic to be instantiated. The *busy* signal indicates that a module that reuses logic is currently operating and that no further information should be passed to it. The *busy* signal is **back propagated** to the *wait* port of all parent modules. The remaining ports to any shader are equivalent to the inputs and outputs for a graphics shader, such as vertex and colour data.

Shaders can range in complexity. This complicates the conversion of a graphics pipeline design for use in a reconfigurable logic environment. There can be a large number of data ports or a large number of operations that can be performed. Graphics often work using matrix operations which can demand a large numbers of resources, as discussed in Chapter 3. It is also necessary to monitor the data being passed into or out of modules, ensuring it is used appropriately. Complex designs quickly translate to large resource use.

5.2.3.4 Vertex shaders

The vertex shader is responsible for moving input vertices within the 3D space based on a transformation matrix. Graphics rendering and computer vision systems use homogeneous co-ordinates, (xw, yw, zw, w), where w is a scalar which allows the point to be moved in the projective space. The vectors that represent the co-ordinate are multiplied by the transform matrices. There are three types of linear transformation that can occur: translation, scaling or rotation [158]. Each of these has an associated matrix. Equation (5.4) is the translation matrix that allows an object to be moved in 3D space.

$$T = \begin{bmatrix} 1.0 & 0.0 & 0.0 & t_x \\ 0.0 & 1.0 & 0.0 & t_y \\ 0.0 & 0.0 & 1.0 & t_z \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \tag{5.4}$$

Equation (5.5) is the scale matrix that allows the size of object to be changed in 3D space.

$$S = \begin{bmatrix} S_x & 0.0 & 0.0 & 0.0 \\ 0.0 & S_y & 0.0 & 0.0 \\ 0.0 & 0.0 & S_z & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \tag{5.5}$$

Equation (5.6) are the rotation matrices that allows the angle of an object to be changed in 3D space. There is a different form for each direction the rotation is to be applied in.

$$\begin{aligned} R_x(\theta) &= \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & cos(\theta) & sin(\theta) & 0.0 \\ 0.0 & -sin(\theta) & cos(\theta) & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \\ R_y(\theta) &= \begin{bmatrix} cos(\theta) & 0.0 & -sin(\theta) & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ sin(\theta) & 0.0 & cos(\theta) & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \\ R_z(\theta) &= \begin{bmatrix} cos(\theta) & -sin(\theta) & 0.0 & 0.0 \\ sin(\theta) & cos(\theta) & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \end{aligned} \tag{5.6}$$

Vertex shaders often perform more than one transformation on incoming vertices. The individual matrices are multiplied together to create a complete *world transformation* matrix.

Chapter 3 demonstrated implementing matrix operations from scalar operations. A design for an $m \times n$ matrix and $n \times 1$ vector multiplier was presented. The same design techniques can also be applied to create matrix-matrix multipliers. These techniques create a set of 'matrix function atoms' for use in hardware design. The greater the number of dimensions of the inputs, the greater the resource use or latency.

The use of transformation matrices significantly increases the flexibility of the shader. Table 5.1 shows metrics for different types of vertex shader. (*Shaders are referred to by numerical reference to the row in the Table.*) Shaders are used to apply the same translation to a vertex.

Two approaches are assessed here: dedicated functions and transformation matrices. For

example, consider a rotation of an object about its center in the Z-plane. This requires three transformation operations: the object is **translated** to the origin of the co-ordinate system; **rotated** about the Z plane and **translated** back to the object's origin. Shader one from Table 5.1 has been made by evaluation of the complete transformation matrix, extracting the functions to apply to x, y, and z and then implementing those individual functions (dedicated function implementation). Shaders two perform a matrix/vector multiplication (transformation matrix implementation). The transformation matrix is loaded by the host.

Table 5.1: Resource requirements and timing analysis for a selection of vertex shaders implementation on an FPGA. Implementations are using half-precision floating-point accuracy.

Module	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs Recoverable by Dense Packing	[C] Estimate of ALMs Unavailable	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
Dedicated Z rotate	2586.2 (0.0)	3614.5 (0.0)	1028.3 (0.0)	0.0 (0.0)	3006 (0)	6480 (0)	8	202.51	199.04
Generic world matrix vector multiplier	838.7 (0.0)	1012.5 (0.0)	175.8 (0.0)	2.0 (0.0)	1134 (0)	1780 (0)	1	214.04	206.48

Constraining shader two to use a world transformation matrix lowers the total number of resources. However, the world transformation matrix must be constructed elsewhere. Once constructed, the world transformation matrix can be stored in on- or off-chip memory to be loaded by the hardware. Alternatively, the shader could be constructed from a number of matrix multiplication elements. The resource use would then be dependent on the number of add and multiply elements required. Implementing the shader as two matrix-matrix multipliers (to multiply the three transformation matrices together) followed by a matrix-vector multiplier would have significant overhead. Consequently, the implementation would become bigger than using the dedicated shader, entry one in Table 5.1.

Using the matrix/vector multiplication method has two benefits. Firstly, there is a much greater comparison between a GPU and the hardware. Secondly, the hardware becomes much more flexible as values for the transformation matrices may be updated by passing new values into the hardware. Table 5.1 also compares the performance of the two methods, demonstrating that the matrix method has a higher throughput.

5.2.3.5 Fragment shaders

Fragment shaders are used after geometric processing. The input is a stream of pixel values and the output is an RGB (colour) value to be written to the memory location associated with the pixel. The colour value includes texturing and lighting. The shaders implemented

Table 5.2: Resource requirements and timing analysis for a selection of fragment shaders implementation on an FPGA. Implementations are using half-precision floating-point accuracy.

Module	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs Recoverable by Dense Packing	[C] Estimate of ALMs Unavailable	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
Flat colour shader	34.5 (34.5)	51.0 (51.0)	16.5 (16.5)	0.0 (0.0)	5 (5)	130 (130)	0	603.5	653.17
Gradient interpolation shader	2731.5 (294.1)	4062.5 (343.2)	1340.0 (49.1)	9.0 (0.0)	2522 (428)	7293 (438)	4	194.93	194.59
Point light shader	21811.5 (8458.1)	28226.0 (7774.7)	7384.1 (249.7)	969.6 (933.1)	13390 (1894)	53568 (13941)	30	124.46	126.1

here are a flat colour, gradient shade and point light source illumination. Computational problems have an associated computational complexity, which is defined as the minimum number of resources required solve a given problem, using the smallest solution from all of the possible algorithms. Examples from graphics processing include tasks such as the flat fragment shader. This has a very low computational complexity (34.5 ALMs in Table 5.2). Conversely lighting tasks have a very high computational complexity (21811.5 ALMs). However, there may be many different algorithms for the same problem. The different algorithms will have a different associated resource cost. To maximise the potential of the FPGA, the algorithm that leads to the lowest computational complexity for a problem should be used, provided it meets all other criteria such as throughput. In an FPGA architecture, resources such as mathematical function blocks can be reused. Doing this wherever possible will reduce computational complexity.

A fragment shader could be as simple as writing a constant RGB value into the pixel locations. Implementing this in hardware requires minimal resources and has a high throughput, in this case limited to the switching speed of the FPGA fabric, Table 5.2. However, basic shaders such as this result in flat images that lack vibrancy and so are not widely used.

Interpolation is a common technique that uses the location of the pixel within the object (relative to the boundaries of the object) to apply more interesting colouring. Shader two in Table 5.2 uses interpolation to mix two colours and achieve a smooth transition from one to the next. Similarly, if an image was being used to texture a triangle, the interpolation data would be used to select the part of the image the pixel represents. Even the simple colour mix performed here requires significantly more resources than applying a flat colour to an object so the performance is much lower.

Lighting is common in modern computer graphics. Introducing lighting calculations has the potential to cause the hardware to grow rapidly. There area number of types of lighting that can be applied, such as point and directional light sources. These cause objects to cast

shadows which change the RGB value of nearby objects. Calculating all these ray paths is resource and time intensive. Shader three in Table 5.2 demonstrates there is an almost eight fold demand in ALMs above the gradient interpolation shader. The increase in calculations has also caused a reduction in the performance of the shader.

It has been shown how the complexity of a shader causes a rapid increase in the number of resources required. The more complex shaders can also suffer a reduction in performance. (Appendix G presents resource and performance metrics for the individual elements of the FPGA-GPU implemented using double- and single-precision floating-point functions.) The increase in precision also increases resource use. Later, this Chapter will discuss whether higher floating-point precision is necessary.

5.2.4 Considerations for designing and implementing the FPGA-GPU

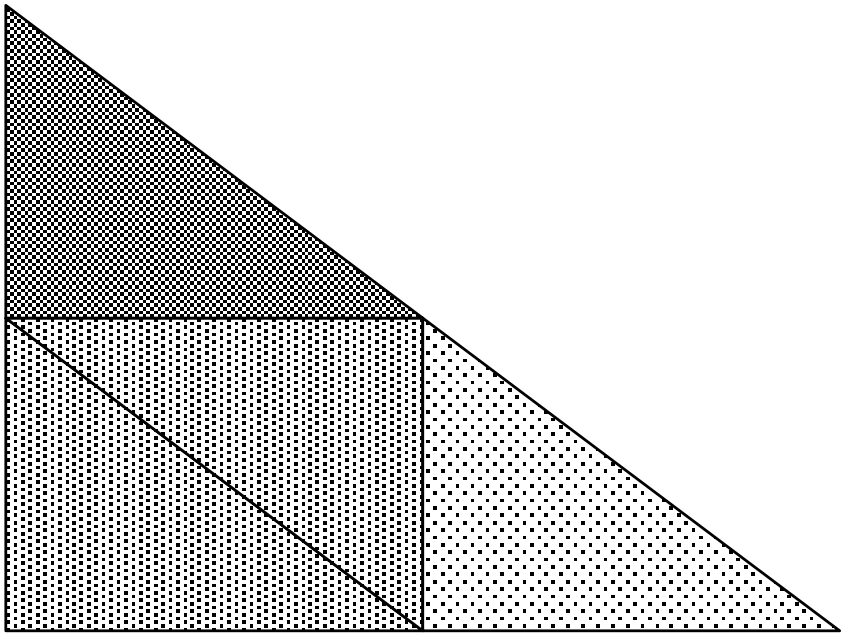
5.2.4.1 Forward rendering

There are two types of rendering that can be performed in a graphics system: deferred and forward. Deferred rendering performs calculations at the end of the processing chain [159, 160]. Deferred rendering tends to be a computationally intensive process as calculations are performed for **every** pixel of an object. This results in a very smooth and lifelike image. Forward rendering moves some of the computation to the beginning of the processing chain (the vertex shader). Calculations are performed on the vertices that define an object. The number of calculations is reduced and they can be performed more slowly saving resources and power; however the image has coarser detailing.

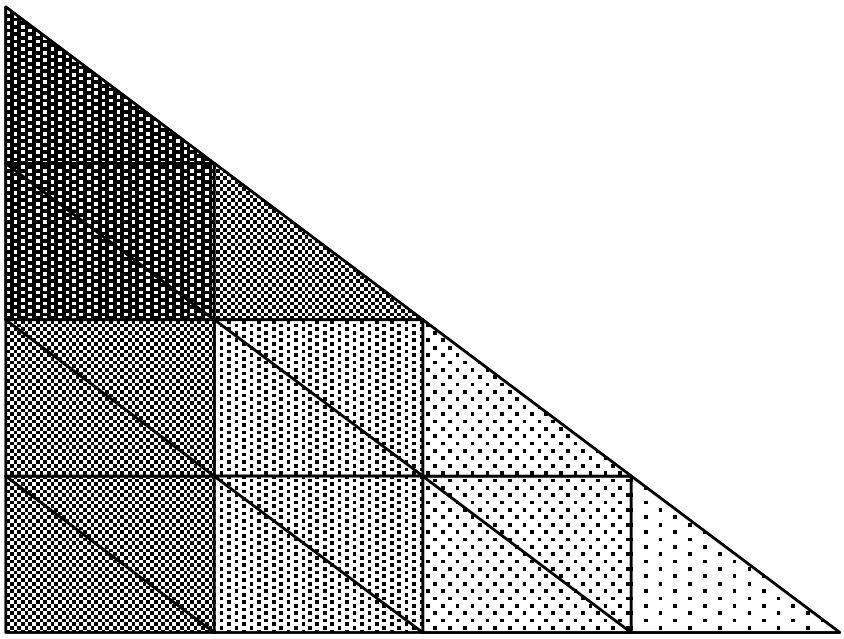
The density of the triangles used to make up an object is important in forward rendering, Figure 5.4. Colours, textures and lighting are applied uniformly across a triangle. Larger triangles will cause big patches of an object to have the same colour applied, shown in Figure 5.4a. Reducing the size of the triangles reduces areas of uniform colour but increases the number of calculations, shown in Figure 5.4b.

Using deferred rendering can lead to large fragment shaders, Table 5.3. The top two rows of the Table are a deferred rendering system; the bottom two rows are a forward rendering implementation. The combined resource use for the vertex and fragment shader using forward rendering is 2,500 fewer Adaptive Logic Modules (ALMs) and 758 fewer registers than the deferred rendering counterparts. Using smaller shader implementations allows parallel instantiation of hardware that can increase throughput by distributing tasks.

Chapter 6 will discuss how dynamic reconfiguration can be used to implement multiple



(a) Low triangle density



(b) High triangle density

Figure 5.4: When textures or lighting are being forward rendered a high primitive density is required for a smooth effect. A low triangle density, 5.4a, produces a blocky texture with sharp transitions between colours. A high triangle density, 5.4b, allows for a smoother transition and more subtle effects.

accelerators in the same device at runtime, so long as they are mutually exclusive. Allowing the FPGA to switch between deferred and forward rendering allows resources to be freed for other hardware accelerators, without stopping the rendering process.

Table 5.3: Comparison of resource requirements and timing analysis for forward and deferred rendering of graphics on an FPGA. Implementations are using half-precision floating-point accuracy.

Module	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs Recoverable by Dense Packing	[C] Estimate of ALMs Unavailable	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
Generic world matrix vector multiplier	838.7 (0.0)	1012.5 (0.0)	175.8 (0.0)	2.0 (0.0)	1134 (0)	1780 (0)	1	214.04	206.48
Point light shader	21811.5 (8458.1)	28226.0 (7774.7)	7384.1 (249.7)	969.6 (933.1)	13390 (1894)	53568 (13941)	30	124.46	126.1
Generic world matrix vector multiplier with forward lighting calculations	12060.5 (251.5)	18372.2 (322.3)	6369.7 (71.0)	58.0 (0.2)	10356 (344)	35391 (539)	18	151.88	157.53
Point light shader using forward lighting calculations	7955.2 (5728.1)	9920.9 (6742.7)	2130.1 (1173.6)	164.5 (159.0)	4101 (1407)	19199 (13533)	13	111.78	112.08

5.2.4.2 Passing data between the vertex and fragment shaders

Values calculated in the vertex shader will be needed in the fragment shader. This can be done with a delay register chain that matches the latency of the rasterizer. However, this is a very resource-costly approach. Alternatively, data can be store in on- or off-chip memory. The amount of memory required for this approach is minimal as only two sets of data need to be stored at any one time. The first set corresponds with the vertices being processed by the rasterizer, the second set corresponds with the vertices waiting to be processed. Once the rasterizer begins to stream out pixel data, the fragment shader loads the associated values from the memory. This results in a method where data is moved efficiently between the two shaders without resort to pipelining.

There are a number of approaches for communicating data between the vertex and fragment shaders. They include registers, distributed memory, block RAM and external memory. To choose which method is to be used, the amount of data and the available FPGA resource must be considered. The decision of which can be used is determined by a number of equations (5.7 and 5.8).

$$D_{bits} \times N_p << R_{ff} \tag{5.7}$$

$$D_{bits} < FPGA_{memory} \tag{5.8}$$

D_{bits} is the total number of bits required to represent the data being moved, N_p is the number of pipeline stages required, R_{ff} is the total number of spare FPGA flip-flops, and

$FPGA_{memory}$ is the total number of spare memory bits left on the FPGA.

A register chain between the source and destination is the sensible choice if the delay (in clock cycles) and the amount of data to be moved is significantly less than the remaining flip-flops on the FPGA equation (5.7). The difference must be significant to ensure the FPGA design still meets timing closure.

If equation (5.7) cannot be satisfied but equation (5.8) can, then the data should be stored in the FPGAs internal memory. Internal memory has a much higher read/write speed than external memory. As a result, the system will not suffer from bottlenecks caused by memory accesses.

In the third case, when both equations (5.7 and 5.8) cannot be satisfied, the only option is to store the data in external memory. As external memory has a significant access time penalty, the data should be separated into frequently and infrequently accessed data. The frequently accessed data is stored internally on the FPGA (where possible) to mitigate the bottleneck.

5.2.5 Designing for system bottlenecks; maximising performance for minimal resource cost

There are a number of limitations to be considered in the FPGA design. Graphics rendering processes are computationally intensive; the more intensive a process, the larger the hardware. Larger shaders also incur a performance cost. The maximum throughput of the system is determined by its slowest part. Additionally, the different shaders work on different amounts of data. The vertex shader only processes vertices (three per triangle), while the fragment shader must process all pixel values. The larger the triangle being rendered, the greater the number of pixel values that must be processed.

Consider a render that transforms vertices by a world matrix and illuminates objects from a point light source. From Tables 5.1 and 5.2 it is seen that the operating frequency of the vertex shader (206.48 MHz) is greater than that of the fragment shader (126.1 MHz). In addition the fragment shader is processing greater numbers of data points than the vertex shader. It is concluded that the vertex shader is performing faster; the fragment shader is creating a bottleneck. Chapter 3 demonstrated that there are many ways in which a matrix/vector function can be implemented, trading-off resource use with performance. When the system's throughput is below the throughput of a performance optimised implementation, using resource optimised implementations is sensible. In some cases, resource optimised

designs will still have a higher throughput than the bottleneck. Furthermore, using resource optimised implementations helps to offset the high resource use from other modules and limits unnecessary resource use. Additionally, bottlenecks can be overcome by implementing multiple computationally intense modules in parallel. Multiplexing between the instantiations increases the throughput of the stage.

These optimisations techniques are only effective until the throughput exceeds the next slowest stage in the system. The new bottleneck must then be evaluated for. Repeating these design optimisations until all the resources have been used will give the best possible performing design.

5.3 Performance of the FPGA implementation compared to embedded GPU devices

Comparisons of performance have been made between the FPGA-GPU implementation and several commercial embedded processors. To compare the performance, the OpenGL render engine was updated to function across a variety of platforms. The render engine was configured to provide an indexed list of vertices which would be subjected to the same vertex transform as the FPGA implementation. Culling and screen refresh were turned off and the number of vertices in the indexed list was increased until the framerate started to drop. This ensured that the vertex shader was the system bottleneck for the embedded processor. Power measurements were obtained for the different devices by measuring the power consumption of the complete system before its idle state. The power consumption was then measured again while the OpenGL load was being performed. The difference in the measurements is taken to be the increase in required power to perform the OpenGL operation. The vertices per second and power consumption of the embedded devices were measured and compared with the throughput and power consumption of the FPGA. Embedded devices chosen for the comparison were an NVIDIA Tegra K1 [161], a ARM Cortex-A9 on the FPGA-SoC (software rendering) [162], an Allwinner A13 using a Mali-400 GPU [163] and an Allwinner A13 using the Cortex-A8 processor (software rendering). Three different configurations for the FPGA were tested, each of which used a different optimisation technique. The results are shown in Table 5.4.

The embedded processor devices (those with ARM cores, and the NVIDIA Tegra) have a much higher total power consumption when compared to running on the FPGA alone. As

OpenGL is a software-based task the embedded processors must also support a complete operating system, as well as the associated I/O and memory; all of this increase the total required system power. The FPGA-GPU implementation benefits greatly from not requiring an operating system or additional memory overhead; instead the ARM core embedded in the device runs a bare metal application to stream vertices into the FPGA-GPU. Therefore, despite the high static power of FPGAs, there is a reduction in total power from the FPGA implementation that results in an increased throughput per unit power. Table 5.4 shows the power consumption of each system before and during running the OpenGL task as a demonstration.

Commercially available embedded GPUs provide a reasonable throughput for a low power and financial cost. The NVIDIA Tegra K1 is the most powerful embedded device tested. It processed the greatest number of vertices per second. However, it also used the most power, over four Watts to process when running at maximum throughput. To normalise performance metrics, the throughput per Watt is calculated. From the efficiency metric the FPGA implementation out-performs all of the embedded GPUs and processors.

The lowest efficiency FPGA configuration provides a six fold efficiency increase over the most efficient embedded GPU (Tegra). The highest throughput FPGA configuration performs almost as many calculations per second as the NVIDIA Tegra K1.

Metrics for an NVIDIA GTX 780 have been included for scaling. Throughput and efficiency data has been extrapolated from [164, 165]. Desktop consumer GPUs have access to a large amount of memory that is used to increase the performance of the device; this is shown from the metrics with pre-loading enabled. However, the FPGA implementation is still more efficient than the NVIDIA GTX 780 device.

5.4 Complete FPGA-GPU implementation

The complete FPGA implementation of the GPU has been assessed. Table 5.5 shows the resource use for each implementation in half-precision floating-point. (Implementations for single-precision floating-point can be seen in Appendix H.) The target device was a low-cost Cyclone V System-on-Chip (SoC) from Intel. The Cyclone V device only provides 110 K Logic Elements (LEs) which limited the size of design that could be implemented. Single-

*Number of vertices and fps given in [164]

**Total power dissipation of device from NVIDIA [165]

†Traditional rendering (streamed data)

‡Rendering with pre-loading vertex data to the GPU

Table 5.4: System power consumption and performance per Watt for all vertex shader implementations.

	Shader	Static Power (W)	Dynamic Power (W)	System Power (W)	Vertices per second	Vertex Rate/W
FPGA-GL	Performance optimised vertex shader	5.0796	0.052	5.1316	199M	3.87G
	Hybrid vertex shader	5.0796	0.037	5.1166	33.3M	895M
	Resource optimised vertex shader	5.0796	0.036	5.1156	10.5M	292M
OpenGL	Cyclone V embedded ARM Cortex-A9 Software Rendering	8.352	1.271	9.623	170k	134k
	Allwinner A13 device using Cortex-A8 Software Rendering	4.44	0.397	4.837	32.73k	77k
	Allwinner A13 device comprising Cortex-A8 CPU with Mali 400 GPU	4.44	0.618	5.058	35M	56.6M
	NVIDIA Tegra K1	5.646	4.704	10.35	215M	45.7M
	NVIDIA GTX 780*	-	-	250**	1.98G†	7.9M
		-	-		56.4G‡	225.6M

and double-precision floating-point implementations required far greater resources.

Graphics processors commonly use half-precision floating-point in order to increase throughput. This technique was used in the FPGA implementation, allowing more complex functionality with higher throughput for lower resource cost. The reduction in precision was not observed to degrade the quality of the output. The screen and VGA protocol introduce a number of quantisation factors, using discrete pixels to draw images and the restriction in number of representable colour levels. A GPU retains some of the flexible characteristics of a GPP and is able to increase and decrease the precision of operations where necessary. This is more complicated in hardware, but still possible. Mixed precision implementations would be interesting to explore in future works.

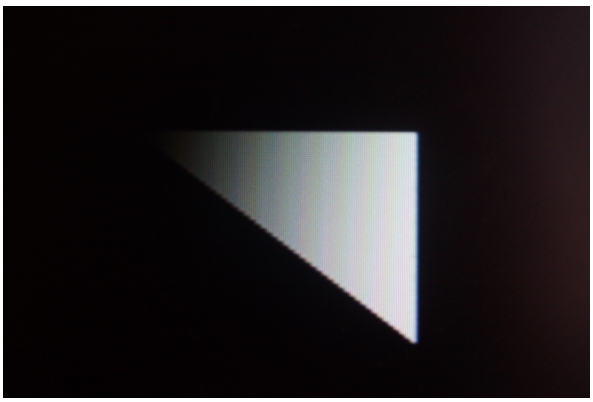


Figure 5.5: Output from the FPGA implementation of a graphics processor. The vertex shader is configured to rotate the object around the object's centre and the fragment shader applies a gradient fill to the object, from black to white.

Table 5.5: Resource requirements and timing analysis for a variety of full graphics processors implemented on an FPGA using the individual components listed earlier. Implementations are using half-precision floating-point accuracy.

Pipeline Type	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs Recoverable by Dense Packing	[C] Estimate of ALMs Unavailable	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85^{circ}C Restricted FMax (MHz)	0^{circ}C Restricted FMax (MHz)
Flat colour fragment shader with generic matrix/vector multiplier vertex shader	17626.5 (380.6)	23351.0 (477.4)	5857.5 (96.8)	133.0 (0.0)	22588 (464)	38036 (323)	13	54.41	57.36
Gradient colour fragment shader with generic matrix/vector multiplier vertex shader	20227.0 (477.8)	27233.0 (564.0)	7121.0 (86.4)	115.0 (0.2)	24961 (609)	45171 (323)	17	54.94	57.82
Point light source colour fragment shader with generic matrix/vector multiplier vertex shader	41317.0 (85.0)	39756.0 (87.5)	387.5 (2.5)	1948.5 (0.0)	36180 (14)	91131 (323)	43	55.71	57.93
Point light source fragment shader with generic matrix/vector multiplier and forward rendered light calculation vertex shader	36777.5 (84.7)	39715.9 (96.0)	3654.9 (11.5)	716.5 (0.3)	36406 (12)	90751 (323)	43	58.02	60.58
Flat colour fragment shader with dedicated Z rotate vertex shader	18211.5 (431.0)	24432.0 (481.2)	6357.5 (50.7)	137.0 (0.5)	22364 (639)	40387 (99)	20	50.81	52.8
Gradient colour fragment shader with dedicated Z rotate vertex shader	20801.5 (516.7)	28078.5 (570.5)	7418.5 (53.8)	141.5 (0.0)	24751 (783)	47543 (99)	24	52.58	55.14

5.5 Summary

In this Chapter, an FPGA implementation of a GPU was presented. Previous literature has used FPGA technology to accelerate parts of graphics processes or replicate a GPU by building a Very Long Instruction Word (VLIW) architecture processor. The implementation in this Chapter differs from these as the entire pipeline has been implemented using floating-point hardware accelerated functions.

The FPGA implementation has been designed to replicate an OpenGL pipeline. A host processor loads vertex values to the hardware which are then operated on to fill a frame buffer. The modules use a standard control interface so that shaders can be easily swapped with each other, similar to how they would be loaded to a GPU at runtime. The next Chapter will discuss using *dynamic reconfiguration* to change FPGA configurations at runtime.

A number of different configurations have been explored. Increasing the functionality (for example adding interpolation or lighting) quickly increases the resource requirements. A number of optimisation techniques were presented and discussed. Using the optimisation

techniques allows computationally intense designs to fit into resource restricted devices - for example the low-cost Intel Cyclone range. Optimisation techniques included trading deferred rendering for forward rendering and exploiting bottlenecks to identify where to implement resource optimised accelerators without compromising overall system throughput. System performance can be enhanced by identifying where bottlenecks occur and implementing multiple modules in parallel, resources permitting.

Comparisons of vertex process rate and efficiency between FPGA and Commercial Off-The-Shelf (COTS) processors was also presented. Similarly priced embedded units were chosen, the most powerful being an NVIDIA Tegra K1. The Tegra was able to process more vertices per second but had a higher power requirement. The power required by the Tegra exceeded that of the FPGA by two orders of magnitude. Measuring performance as throughput per Watt demonstrated that the FPGA out-performed the embedded GPUs. The resource optimised FPGA had a six fold efficiency gain over the Tegra.

Implementation of a graphics processor on an FPGA demonstrated how designs for hardware-based mathematical functions, Chapters 3 and 4, can be integrated into a complete system.

Chapter 6 will present changing the FPGA's configuration at runtime. Discussions will be presented regarding changing the nature of the graphics rendering or allow a different task to be accelerated. Chapter 6 will further discuss context switching methods for hardware accelerators.

Chapter 6

Dynamic Task Allocation and Context Switching

This research has presented work regarding flexible, reconfigurable logic used as hardware accelerators. A key topic this Thesis wishes to address is the benefits and limitations of using hardware acceleration in a **dynamically reconfigurable** environment. An environment that is dynamically reconfigurable can change its functionality at runtime. This is similar to a processor: the task being performed can be changed quickly. Unlike a processor, hardware must be physically altered to change the task being performed.

This Chapter will consider a number of concepts that are important for dynamic reconfiguration in heterogeneous environments. These consideration are the different types of dynamic reconfiguration, examples of architectures, and context switchable hardware accelerators.

6.1 Dynamic reconfiguration

Dynamic reconfigurability has the potential to increase the flexibility of hardware. Dynamic reconfiguration has great appeal as it would allow both the hardware's speed and efficiency and the flexibility of software. Early (pre turn of the century) uses of FPGAs to provide dynamically reconfigurable platforms are discussed by Butts, [166]. The work by Butts highlights the use of large arrays of FPGAs and Programmable Interconnect Arrays (PIA) in reconfigurable computers. It is asserted that reconfigurable computers have fewer limitations than conventional CPUs, such as parallelism. Early FPGAs, such as the Xilinx XC6200, supported dynamic reconfiguration to help compensate for their low resource count. There are a number of publications that examine uses cases and the performance of devices such

as the XC6200 [167, 168]. Although the research is not extensive, there is initial analysis is performed for using the devices in a partially reconfigurable manner. McKay *et. al.* present methods for transforming logic cells (such as AND or OR functions) into wiring cells, and use this to evaluate circuits including adders, multipliers and FIR filters [167]. However, as commercial FPGAs increased in size, the partial reconfiguration support was dropped. Modern FPGAs are starting to re-introduce support for dynamic reconfiguration to increase the flexibility of the device by allowing architectural changes at run-time. Xilinx presented dynamic reconfiguration at the Field Programmable Logic (FPL) conference in 2006 [169], and Altera released documentation in 2010 [170]. Consequently, the technology is still in its infancy. There are two types of dynamic reconfiguration - full and partial.

6.1.1 Full reconfiguration

Full dynamic reconfiguration completely erases the configuration of an FPGA and loads a new configuration in its place. This is the more simple of the two reconfiguration types. It does not need to separate the floor plan of the FPGA or provide clock wrappers and standard interfaces to regions.

The full reconfiguration is not dissimilar to loading an FPGA configuration on power up. The main advantage is that this can be performed at any point during the system's execution without re-writing the configuration memory and power cycling the device. Full reconfiguration requires a host capable of configuring the FPGA. In a modern FPGA-SoC device, there is always a host within the FPGA that can perform these tasks.

6.1.2 Partial reconfiguration

Partial dynamic reconfiguration targets isolated parts of the FPGA, allowing a change in configuration while the rest of the design still operates.

Partial reconfiguration presents a more complicated design problem. In an example application, sections of a design are edited at runtime to change the type of a filter in a signal processing chain. In order to accomplish this, sections subject to partial reconfiguration must be identified at the design stage. The reconfigurable parts of the design must be isolated from the static logic. Additionally, each reconfigurable variation must have a common interface. Furthermore, control signals, such as clocks and resets, must be moved into a known state to prevent toggling which could cause configuration errors. These control signals must not impede function of the rest of the design.

```
1. Create the base revision. This is the configuration that will be
   loaded into the FPGA at startup.
2. Identify the parts of the base revision that are subject to partial
   reconfiguration.
3. Instruct the tool that these locations must be locked and the regions
   are partially reconfigurable.
4. Create new revisions that keep the static logic the same but detail
   the new logic for the reconfigurable regions.
5. Ensure all revisions are compiled.
6. Generate full and partial programming streams for the base and
   partially reconfigurable revisions.
```

Listing 6.1: Design flow for partial reconfiguration

Listing 6.1 shows the steps that must be performed to set up a partially reconfigurable design using an Integrated Design Environment (IDE). The Listing does not cover specific design nuances.

In this research partial reconfiguration was achieved using an Intel Cyclone V SoC. Designs in which partial reconfiguration was enabled also required:

- A freeze wrapper that controls the clock and reset signals for the reconfigurable region
- A partial reconfiguration controller
- A host to pass the reconfiguration files to the reconfiguration controller

Note: For the Cyclone V SoC device there was a silicon fault preventing the internal reconfiguration controller from working - as confirmed by Intel FPGA. This was circumvented by using a controller designed for the Stratix series of devices. The Intellectual Property (IP) had to be edited to work internally on a Cyclone V device.

6.1.3 Continuous end-to-end data flow

Some applications require the continuous processing of data. For example, data may need some pre-processing before being used by a processor. Real-world data is likely to be continuous and subject to change. The type of data processing required may need to change. Alternatively, the user may wish to change the type of processing being applied to a signal.

Figure 6.1 shows a potential hardware arrangement for processing a signal. In this case each reconfigurable processing region has a bypass stage. The bypass serves two purposes:

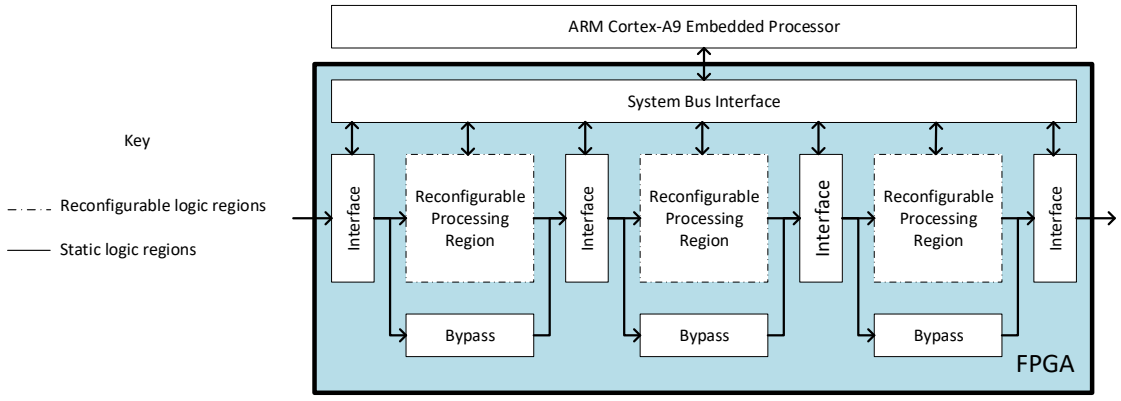


Figure 6.1: An architecture for reconfigurable hardware that accepts a signal and applies a series of functions to it. Each function region can be either bypassed or included in the design depending on the number of processes required. Multiplexors must be used to swap between the different data paths.

maintaining data flow while no transform is to be applied, and providing a pathway for data while a region undergoes reconfiguration. When the system broadcasts a message that the configuration of a region must be changed, data is diverted through the bypass region. This ensures the data stream is maintained.

Bypass regions are easy to implement and require few resources. However, if data must be kept continuous, there is also the potential that the data must always be processed. Figure 6.2 presents an adaptation that increases the number of reconfigurable regions. When a message to reconfigure a region is broadcast, the system identifies which block of a pair is currently not being used. The new configuration is then loaded into the empty block and the data path is switched once reconfiguration is complete. The old configuration can then be erased once any remaining data in that process block has left. This ensures there is never a drop in the output.

6.2 Context switchable hardware

In a heterogeneous system, FPGA fabric would be used as a reconfigurable hardware accelerator, as shown in Chapter 5.

Figure 6.3 shows an FPGA broken into a set of regions. Each region can support a hardware accelerator. The location of each accelerator is determined by the processor based on the current system demands. This allows the processor to make the most optimal use of the flexible hardware by ensuring the most useful accelerators are always implemented.

Interfacing the processor with the hardware adds a number of design complexities. The

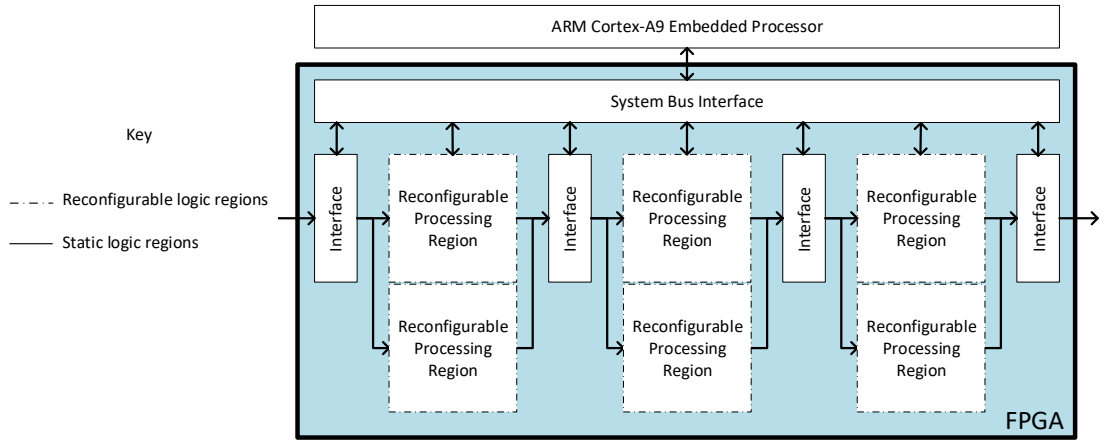


Figure 6.2: Two reconfigurable regions can be placed back to back, ensuring that data can always be subject to processing even while reconfiguration occurs. Multiplexors must be used to swap between the different data paths.

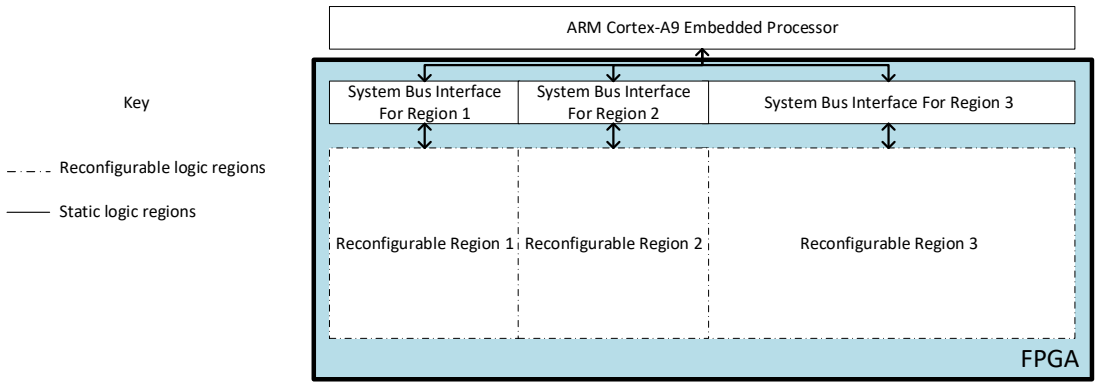


Figure 6.3: The floor space of an FPGA may be split into a number of sections. These sections can either be all the same size or a variety of sizes, depending on the designer. Each section can then be used to load a hardware accelerator.

processor and the hardware are unlikely to be in the same clock domain. Consideration must be given to what happens to data in the accelerator if a processor changes task; what properties determine how important an accelerator is compared to other accelerators; and how to deal with accelerators that become fragmented over the FPGA, causing sub-optimal use of the floorspace and preventing new accelerators from being placed.

Modern processors tend to operate in the region of hundreds of MHz if not GHz; FPGA fabric is typically limited to a few hundred MHz. The difference in operating clocking frequencies creates a barrier between the two that data must cross without being lost. Crossing over a clock domain can be achieved in several ways [171]. The simplest way is to create a synchroniser chain from a series of flip-flops. The incoming signal comes from the original clock domain whereas the clock for the flip-flop chain is provided by the destination domain.

Synchroniser chains can be constructed from any number of flip-flops (greater than one). Synchroniser chains are relatively basic methods of moving data across a clock domain but have associated pitfalls. Other strategies include the use of muxes or dual clock First-In, First-Out (FIFO) buffers.

In order to handle multiple tasks, processors perform context switching. Context switching allows a processor to change task by storing current information, perform a new task and then return to the original task as though there was no interruption. Context switching in a processor is relatively straightforward. There is usually plenty of memory for storing register contents. If the processor is using hardware acceleration, part of the information related to a task is currently stored in the hardware. Consideration must be made as what information is hardware accelerated, to mitigate data perturbation effects from context switching. Alternatively, the processor could wait for the hardware to run to completion before performing the switch. However, this adds latency to the design. This Chapter will present a novel method for performing context switching in hardware accelerators.

The size of hardware accelerators can vary hugely depending on the task to be accelerated. (In Chapters 3, 4 and 5, the Thesis has presented accelerators that range from single mathematical operations to complete systems.) An FPGA has a limited amount of floor space in which to implement the accelerators. Partial reconfiguration requires a continuous area in the FPGA. Constantly changing between accelerators of different sizes may cause fragmentation in the FPGA. Context switchable hardware allows de-fragmentation (discussed in the next Section) to restore continuous FPGA resources.

Careful selection of which elements of the reconfigurable hardware are context switchable significantly reduces resource overhead and latency by creating shorter scan chains - this is termed 'partial extraction'. By ensuring context switches only occur at certain times, for instance only after the entire frame in a convolution kernel has been processed and not during the frame, limits the number of registers whose states require saving. Registers that contain the current state of a state machine, or control flow registers are likely to need to be saved; whereas registers in a data pipeline can be ignored as the data in these get flushed through with a control line being used to indicate if the output data is valid or not. The selection of hardware checkpoints for reconfigurable systems was discussed by Bourge *et. al.*, [172], where methods are presented to analyse a design and extract the checkpoints based on latency requirements. A finite state machine identifies a number of checkpoints from the

design and a greedy heuristic is used to minimise the NP-Complete* problem and obtain a set of checkpoints the minimal area overhead.

Determining which accelerator should be loaded into which area of the FPGA presents a significant management task. The embedded host could be running several of tasks, particularly if it is capable of multi-core or multi-thread processing. Additionally, there may be more than one accelerator for a given task. Calculating which accelerators should take precedence requires the system to rank tasks based on importance. Ranking tasks automatically could lead to situations where the system prioritises tasks incorrectly. Instead, having a method where the user can indicate the priority of a task upon launch would give far greater control to the user, limiting the potential for the incorrect tasks to become prioritised. Tasks that have the same level of priority associated with them when launched could still have the potential for the 'wrong' task to be accelerated as the system manager would have to deduce which are the 'best' tasks to accelerated. Deducing the 'best' accelerator could be based on the order the tasks were executed in, or the number of FPGA resources available. If a task that was executed first requires an accelerator that cannot fit, the manager should implement a smaller accelerator for a later task.

6.2.1 De-fragmenting hardware accelerators

There are situations where certain hardware accelerators require more than one contiguous reconfigurable region. Fragmentation of hardware accelerators across the FPGA's floor space could result in inefficient use of the FPGA. A method to move the hardware accelerators around the fabric of the FPGA allows de-fragmentation. Simply moving an accelerator can cause issues: data could be lost or calculations may have to be repeated. This either adds latency or corrupts operations.

In this Section a new method to replicate context switching in hardware is proposed. The method uses standard D-type flip-flop with additional ports. The additional ports allow data to be scanned in to and out of the flip-flop. The proposed flip-flop is termed a 'pre-emptible flip-flop', shown in Figure 6.4. The scan_in and scan_out ports work like a scan chain. Data is passed in and out of the flip-flop while the hardware is not functioning. The state port controls the operation of the flip-flop, switching between the running and save/load modes.

Netlist views of a D-type and the proposed pre-emptible flip-flop are shown in Figures 6.5 and 6.6 respectively. It can be seen the pre-emptible flip-flop requires an additional four

*NP-Complete problems are in the Nondeterministic Polynomial time (NP) set of all decisions problems whose solutions can be verified in polynomial time.

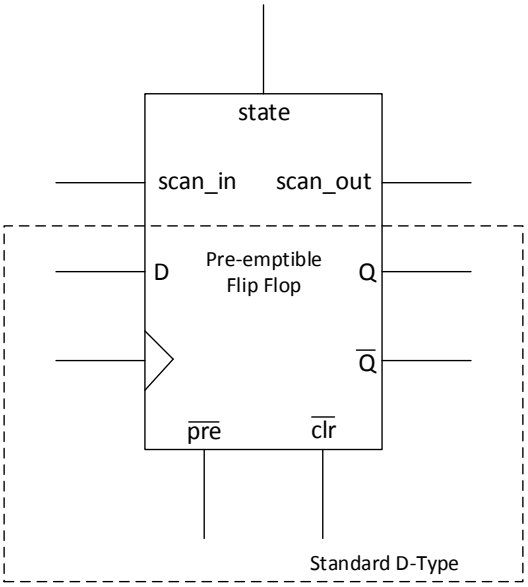


Figure 6.4: The pre-emptible flip-flop builds on the basic D-type flip-flop common in modern logic circuits. Additional ports are added that give control to the user to add in or remove data while the circuit is not functioning. A state port allows the flip-flop to toggle between running and save/load mode.

muxes. The design for the pre-emptible flip-flop has been re-developed from the originally proposed design in [173]. The main benefit of this new design is that the resource overhead has been reduced.

The pre-emptible flip-flop adds minimal logic overhead. Table 6.1 shows the resources for the new flip-flop. The fitting tool reports an increase of **half** an Adaptive Logic Module (ALM) and **one** additional Adaptive Look-Up Table (ALUT) for the four muxes. Table 6.1 also shows that the maximum operating frequency for the pre-emptible flip-flop is $> 500\,\text{MHz}$, restricted by the switching speed of the FPGA's fabric.

Table 6.1: Comparisons of the resource requirements and timing analysis between a standard D-Type flip flop and the augmented pre-emptive D-Type flip flop used for context switching.

Module	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs recoverable by Dense Packing	[C] Estimate of ALMs unavailable	Combinational ALUTs	Dedicated logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
Regular D-type flip flop	2.5 (2.5)	2.5 (2.5)	0.0 (0.0)	0.0 (0.0)	5 (5)	1 (1)	0	-	-
Pre-emptible D-type flip flop	3.0 (3.0)	3.5 (3.5)	0.5 (0.5)	0.0 (0.0)	6 (6)	1 (1)	0	530.79	542.59

6.2.2 Controlling context switching in hardware

Chapter 2 presented existing methods of context switching. Context switching can be performed using the operating system of a processor to move register contents around. This

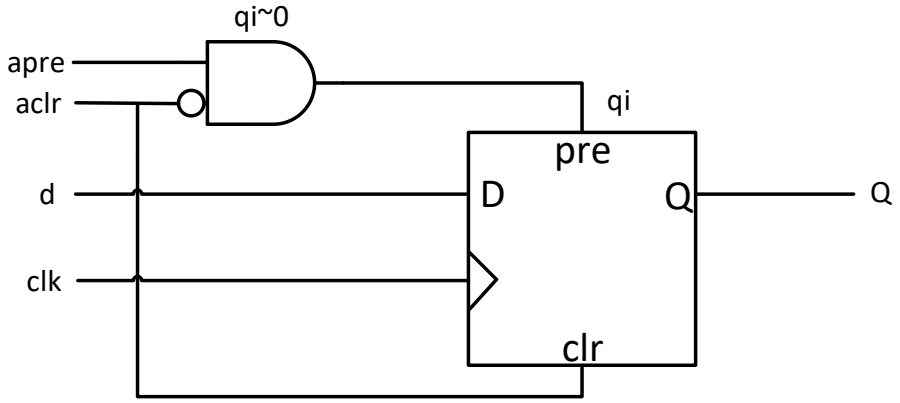


Figure 6.5: The netlist view of a standard D-type flip-flop as described by the Cadence tool suite for the synthesis of hardware designs.

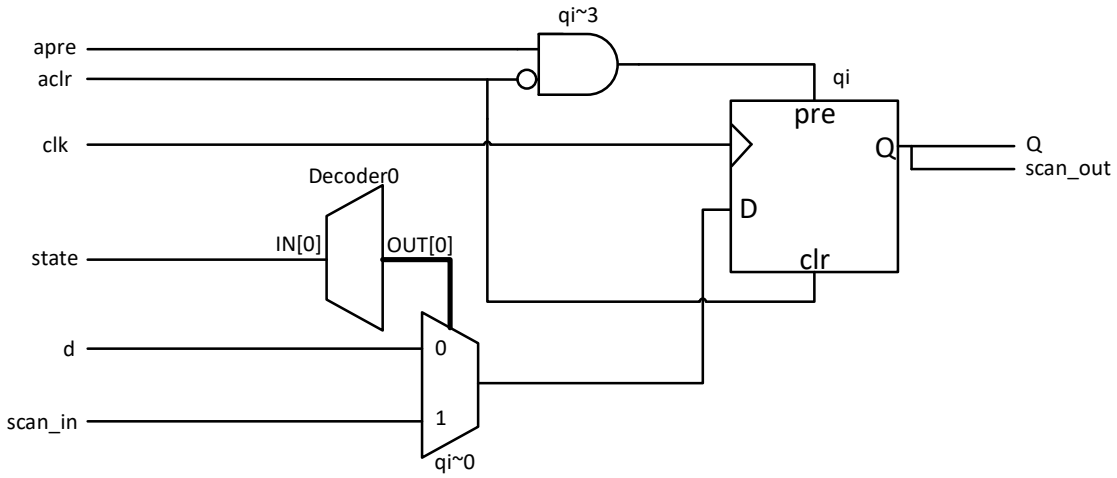


Figure 6.6: The netlist view of a pre-emptible flip-flop. The state line controls whether the flip-flop is being used for normal operation or whether its contents is being loaded/saved. Realising the flip-flop with the additional ports requires only a small overhead in terms of logic components.

is extremely time-consuming, requiring thousands of clock cycles to complete. The original version of the hardware context switching in this research used a similar method [173]. The Hard Processor System (HPS) on the FPGA managed the load and store operations. Data was chained through the pre-emptible flip-flops and passed to the HPS which saved the data to accessible memory. Similarly, a load used the HPS to read the data from memory and feed it back into the FPGA. However, this method was slow and added additional overhead and complexity to the system. The use of the HPS also prevented the processor from executing additional tasks while the hardware was context switched. Implementing dedicated hardware based units to manage context switching significantly decreases execution time [98, 99].

To overcome these limitations, a revised context switching controller is proposed. The processor issues a context switch request to the control bus. The request details whether

a load or a store is required, and the length of the data. Once the processor receives an acknowledgement from the hardware it continues executing other routines. The manager provides a wait signal to ensure it is not interrupted while a save or load operation is being executed.

The manager directly interfaces to on- or off-chip memory. Internal memory has the benefit of lower latency access but requires additional FPGA resources. For the purposes of testing, the context switching manager was implemented using external Synchronous Dynamic Random Access Memory (SDRAM).

Table 6.2: Resource requirements and timing analysis for two different FPGA based controllers to enable hardware based context switching. Arrangments are either in serial or parallel.

Module	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs recoverable by Dense Packing	[C] Estimate of ALMs unavailable	Combinational ALUTs	Dedicated logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
Parallel context switching controller	266.0 (125.8)	302.5 (143.6)	43.0 (24.2)	6.5 (6.5)	392 (172)	354 (119)	0	94.05	96.67
Serial context switching controller	325.5 (180.8)	354.0 (191.9)	29.0 (11.6)	0.5 (0.5)	472 (241)	411 (180)	0	106.64	109.63

The manager interfaces to the FPGA with either a serial or a parallel bus, shown in Figures 6.7 and 6.8 respectively. Table 6.2 provides metrics for both interfaces. The serial arrangement requires additional control logic to construct the memory packets. Data is scanned into the manager one bit at a time. It is loaded into a shift register until the entire packet is constructed. Similarly, when loading the state of registers from memory, the data is read as a packet and loaded back into the FPGA one bit at a time. Although the serial interface is capable of operating at a higher frequency, the parallel interface allows a higher throughput by reducing the latency. The parallel interface matches the data width of the memory device. One clock cycle can be used to read or write a full word from or to the FPGA.

The scan chain poses an important architectural consideration. High fan-out nets have a high resource costs and make meeting timing closure difficult. Partial extraction methods, discussed in Section 6.2, help to reduce the number of end-points for the net to reduce routing demands and make meeting timing closure easier. Additionally, having the scan chain interface as part of the FPGA primitives, with a dedicated routing layer in the device, would make implementing the scan chain easier by not requiring standard routing resources.

Without special resources built-in, the architecture of the design must be carefully considered.

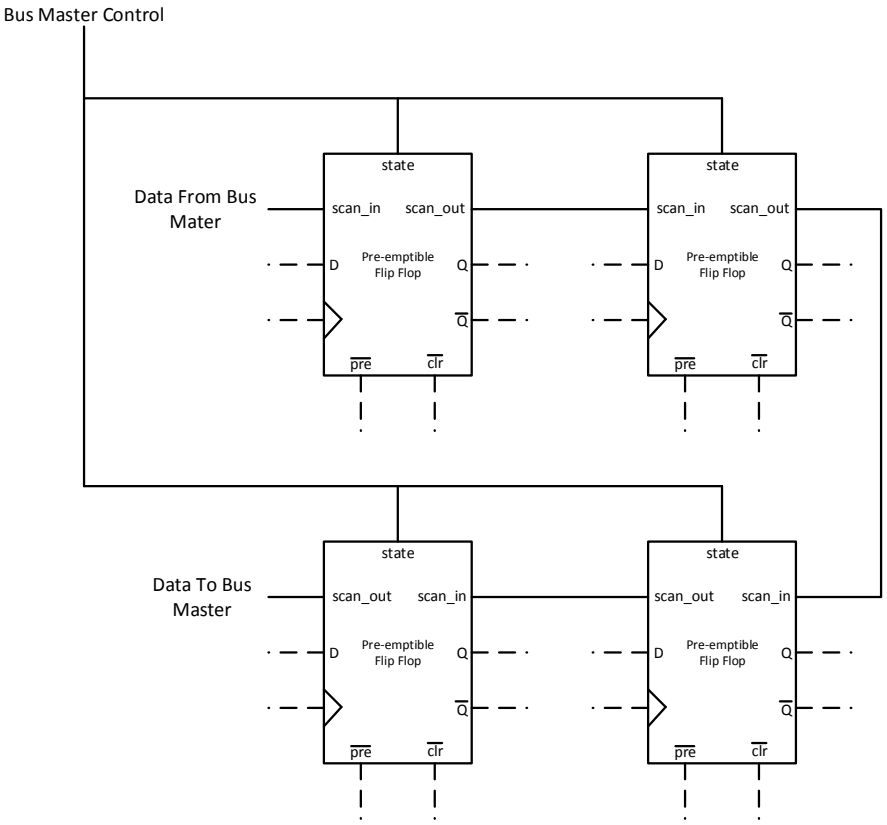


Figure 6.7: To control the saving and loading of context from the pre-emptible flip-flops, a master is implemented. The flip-flops are arranged in a serial pattern.

6.2.3 Effects of using pre-emptible flip-flops on resources and performance

The pre-emptuble flip-flop requires four additional muxes. While these may be available in a standard ALM, it is not guaranteed that the fitter will use muxes from the ALM that contain the normal D-type. The fitter will attempt to minimise the path delay between all elements to balance area against performance. This can result in muxes from other ALMs being required to implement the pre-emptible flip-flop design.

To analyse the effect on resource consumption and performance from using the pre-emptible flip-flop, a number of hardware designs were compiled and analysed. Each design had an increasing number of flip-flops. When a parallel scan chain is implemented, the number of flip-flops can be increased to provide the padding flip-flops, ensuring each chain is the same length. This is shown on Figures 6.9 to 6.11 by the multiple star markers for single quantities of flip-flops. Each compilation was performed a number of times using different fitter seeds to obtain a representative result. The number of registers used for a design with pre-emptible flip-flops is unchanged, shown in Figure 6.9.

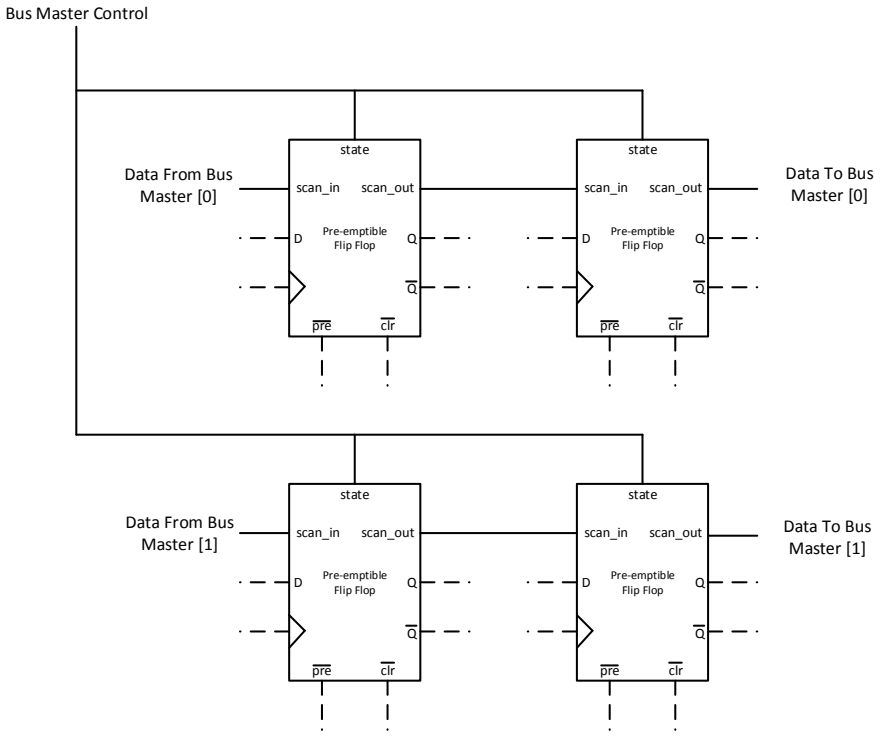


Figure 6.8: Alternatively the pre-emptible flip-flops can be arranged in parallel, where the number of parallel chains can be determined by the data width of the memory device.

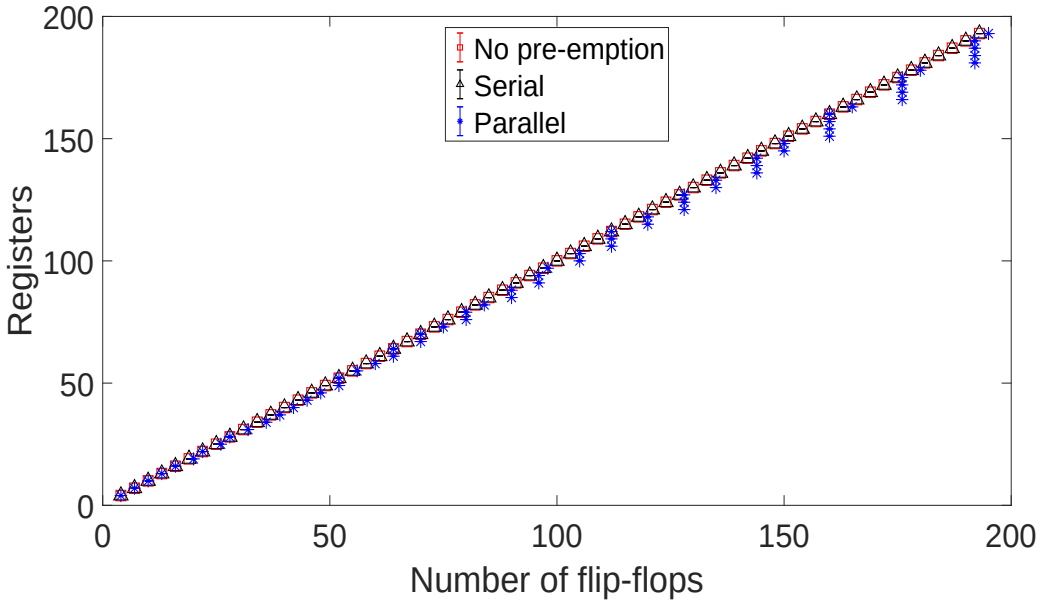


Figure 6.9: Registers used for implementing designs of increasing size before and after pre-emptible flip-flops are inserted. The registers requirements from adding pre-emption, arranged in either serial or parallel, is given by the black and blue markers respectively.

The maximum operating frequency, f_{max}, of a design varies with its size, although pipelining can be used to reduce the routing delay at the expense of requiring additional resources. Figure 6.10 shows that using pre-emption has a far smaller effect on the operating frequency

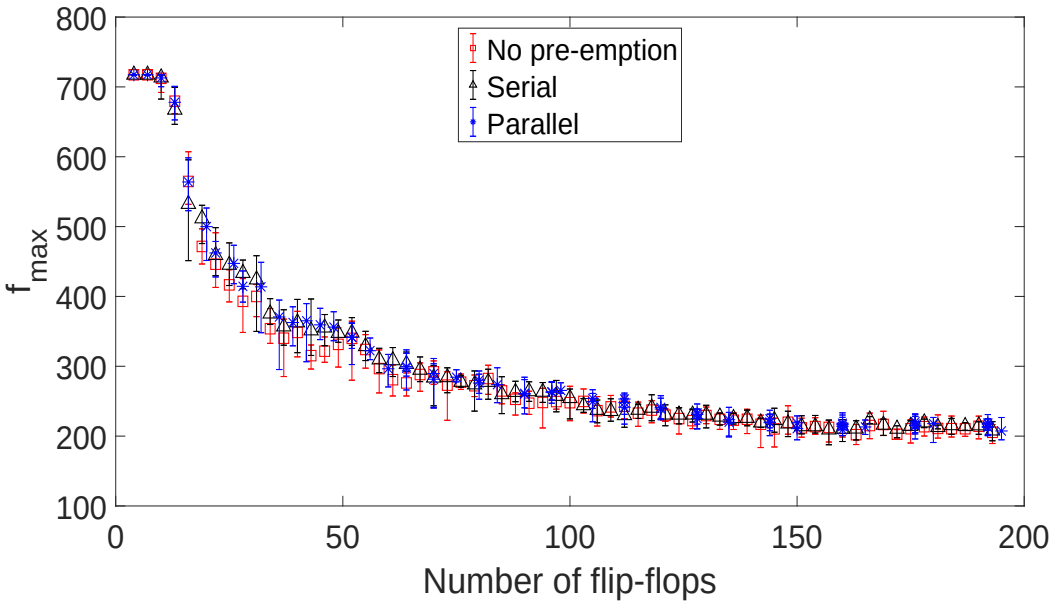


Figure 6.10: f_{max} of designs of increasing size before and after pre-emptible flip-flops are inserted. The f_{max} of the design after adding pre-emption, arranged in either serial or parallel, is given by the black and blue markers respectively. Timing data given by TimeQuest.

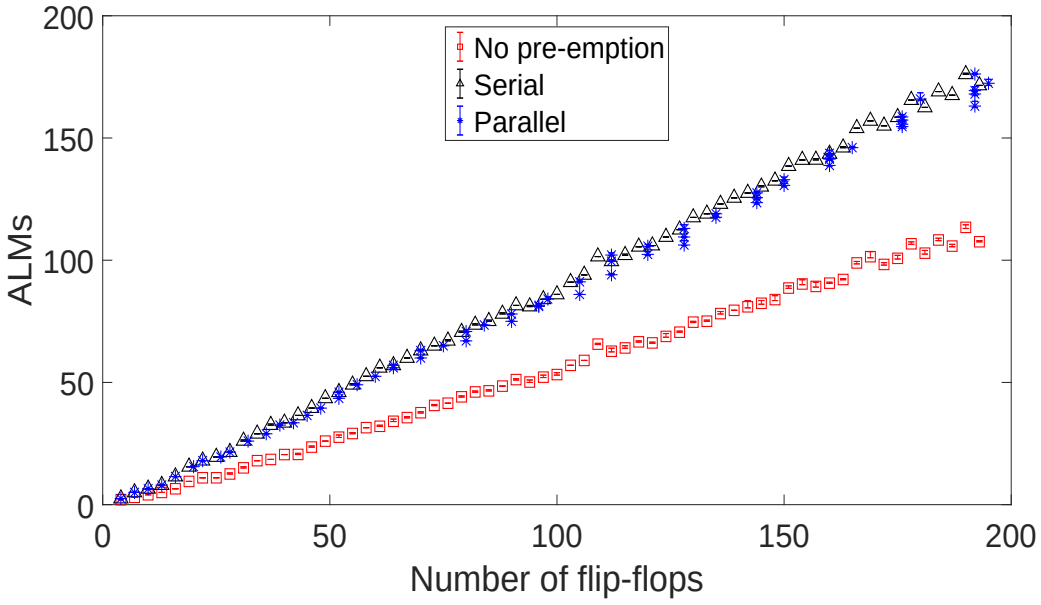


Figure 6.11: ALMs used for implementing designs of increasing size before and after pre-emptible flip-flops are inserted. The ALM requirements from adding pre-emption, arranged in either serial or parallel, is given by the black and blue markers respectively.

than changing the fitter seed. It is concluded that the pre-emptible flip-flop has a negligible f_{max} penalty.

The biggest impact of the pre-emptible flip-flops is the number of ALMs required, shown in Figure 6.11. The additional muxes may not be available in the original ALM. If they

are not available the fitter uses other ALMs on the device. From Figure 6.11, it can be seen that the increase in ALMs needed as the number of flip-flops increases is linear. For a given number of flip-flops, the equivalent pre-emptible design requires approximately 1.6 times more ALMs. Some fluctuations are seen depending on whether the scan chain is serial or parallel, and with widths of registers. This is to be expected since some designs result in register sizes that do not fit inside an ALM, so the fitter compensates.

6.2.4 Including pre-emptible resources in hardware designs

Synthesising a behavioural level design to hardware results in the synthesis engine using standard D-type cells. Replacing the original D-type flip-flops with the new cell can be done at gate level by hand. This is a difficult process that is prone to error. Alternatively, the process was automated.

The automated process works on gate level Verilog and could be included as part of the synthesis chain. In this research, the tools have been run separately as the synthesis toolchains (Intel) are manufacturer protected IP. Further, most synthesis tools prevent access to the gate level Verilog files. The research had access to the Cadence tool suite has allowed the generation of gate level Verilog files that can be accessed and edited. The flow for automating the replacement of the flip-flops is given in Listing 6.2.

The program for automated conversion accepts a user flag to switch between a serial and a parallel implementation. The user can specify the memory width for the parallel interface. For a parallel implementation the program will automatically arrange the flip-flops to produce scan chains with the minimum number of additional padding registers. Padding registers are necessary to ensure each scan chain in the context switchable design is the same length. The maximum number of padding registers can never exceed $datawidth - 1$.

```
1. Synthesise behavioural level HDL to gate level HDL.
2. Read gate level HDL file.
3. Parse the file to identify the module to be adapted and the number of
   flip-flops requiring replacement.
4. Update the port list of the module to include the new scan_out,
   scan_in, and state ports.
5. Update the local declarations to include the connections between the
   scan ports of the pre-emptive flip-flops.
6. Replace each D-type flip-flop with its pre-emptive equivalent.
7. Connect together the scan chain.
8. Re-synthesis the new gate level file. This can now be included in the
   design just as any behavioural level description file can be.
```

Listing 6.2: The pseudocode for automating the replacement of the standard D-type flip-flops with the new pre-emptible flip-flop.

The system has been implemented and tested. Designs for hardware were converted from behavioural level Verilog to gate level Verilog using Genus from Cadence. The Cadence standard design for a D-type was then replaced with the pre-emptible flip-flop using the automated method. The resulting file was then included in a Quartus design. The embedded ARM processor on the Cyclone V SoC loaded data into the hardware and made context switch requests. Once the context switch had finished, the ARM core then probed the hardware to ensure the data had been removed from the system. Context was then loaded back into the system and the ARM core confirmed operation by comparing the result from the hardware with the expected result.

To increase the rigour of the test, dynamic reconfiguration was also used. Full reconfiguration was performed after data was context switched out. This ensured no erroneous data was causing the context switching system to appear to be working. It also demonstrated the possibility of saving the context of an FPGA configuration before dynamically reassigning the hardware accelerators. The original task can be returned to as though there had been no interruption.

Partial reconfiguration demonstrated moving a configuration on the FPGA while the rest of the FPGA was functioning. The context was saved to memory before the configuration was moved from one area of the FPGA to another. The context was then loaded into the configuration in the new region on the FPGA and the ARM core probed the result. All tests validated the operation of the pre-emptible flip-flop, the pre-emption manager, and the

ability to de-fragment or reassign hardware accelerators in FPGA fabric.

Table 6.3: Resource requirements and timing analysis for a complete example system. The system includes the control master for loading and saving the state of flip-flops, an accelerator, the hard processor system, and the required peripheral logic.

System	ALMs	Registers	f_{max} (MHz)
Serial bus master in isolation	475	327	151
Parallel bus master in isolation	409	267	178
Adder in isolation	49	26	309
Adder with pre-emptible flip-flop in serial arrangement in isolation	49	44	365
Adder with pre-emptible flip-flop in parallel arrangement in isolation	52	45	326
Non-context switchable adder with HPS	6,605	5,113	67
Context switchable adder with serial interface with HPS	7,204	5,730	60
Context switchable adder with parallel interface with HPS	7,130	5,422	60

Table 6.3 gives metrics for the individual modules and the complete design. As can be seen, the overhead from adding the pre-emption logic and control is small compared to including the hard processor and its peripherals. It has been concluded that the additional system requirements for pre-emption are negligible.

6.2.5 Pre-empting hard IP blocks

This Chapter has discussed replacing D-type flip-flops with pre-emptible flip-flops. However, FPGAs contain a number of IP blocks designed to improve the performance of the device. Scanning data in and out of these hardware blocks poses further challenges. The use of IP blocks varies based on the device and complete design, and is determined by the compilation tool. While it is possible to prevent fitting tools from using Digital Signal Processing (DSP) blocks or memory cells, this often reduces performance or increases cost (As demonstrated with the design of a floating-point multiplier in Chapter 3). Instead, the IP blocks can be included in the scan chain. If a vendor provide scan chain ports into and out of these IP blocks then the system will work identically to before. An alternative method for some IP blocks is to add a soft logic wrapper. The wrapper is used around memory blocks to provide read and write capability when a context switch is requested.

It is important that the pre-emption insertion tool knows the length of the scan chain through the IP blocks. This information can then be included in the calculations for the

lengths of other scan chains, ensuring the pre-emptible flip-flop scan chain and the IP block scan chain are balanced.

There are alternative methods for including blocks such as memory and DSP. The contents of memory are easy to save and restore. The location and size of used memory must be extracted. This information can be passed to the hardware manager which can connect to memory blocks to execute read and write operations. DSP blocks present more of a challenge. These require a pre-load to restore context. Performing a pre-load requires the system to save previous entries that have been sent to the DSP, on a rolling basis. When a context save is requested this information must be saved. When a context load is requested, this information is clocked back into the DSP, restoring its original state, before the hardware is run.

6.3 On-line compilation and configuration of reconfigurable devices

Being able to perform on-line synthesis and compilation would allow reconfigurable devices to be altered at run-time without pre-compiling source code. Compilation time, particularly from place and route operations, makes this extremely difficult. Place and route algorithms take a synthesised HDL file and iteratively place the components onto the fabric of the target device until the design constraints are satisfied. The development of an efficient, run-time, place and route procedure for reconfigurable devices is far from reality. However, it is possible to break down this problem into smaller, sub-problems and create systems that can approximate on-line synthesis of reconfigurable logic.

Figure 6.12 shows the generic view of the floor plan of an FPGA divided into identical regions of FPGA resources. Similar approaches have been explored on Xilinx technology in the past [174], but the implementation in this research is targeting Intel FPGA technology. The homogeneity of FPGAs makes creating regions of identical resources possible. Each has a standard interface to its neighbouring regions. Each region is also connected to the embedded processor on another interface bus, allowing the processor to communicate with all areas of the reconfigurable fabric. A reconfiguration controller loads configurations into these regions on the FPGA. In some FPGAs, such as Arria 10 SoC or Stratix 10 SoC, there is a reconfiguration controller implemented in the silicon of the device.

The island style reconfigurable solution, shown in Figure 6.12, was demonstrated using

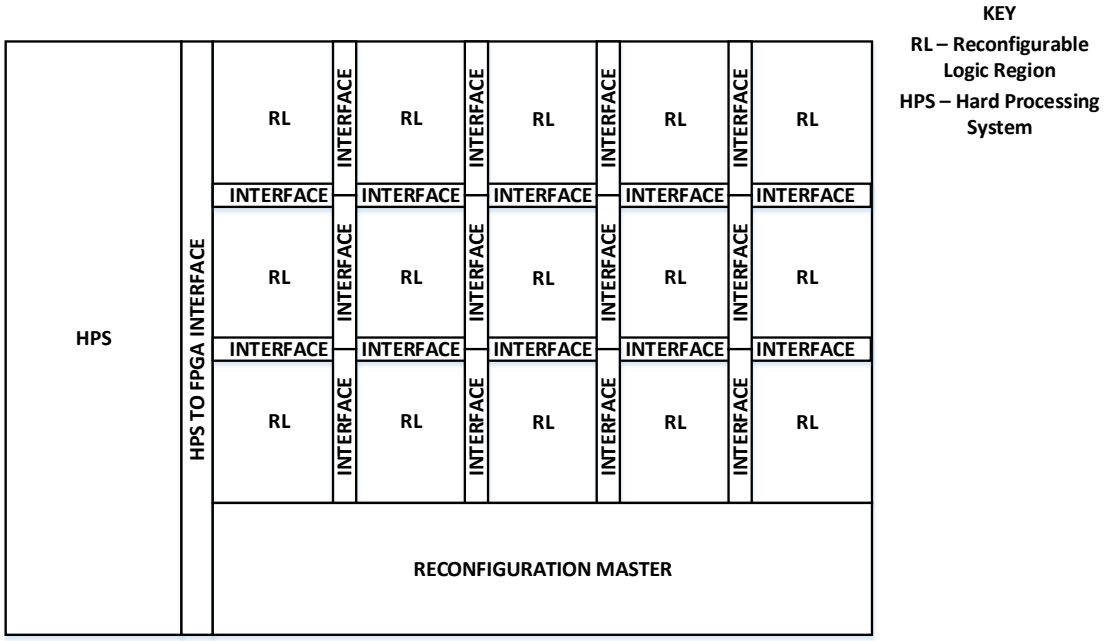


Figure 6.12: The FPGA-SoC device can be divided into a set of reconfigurable regions with common interfaces. Each region can be configured with a hardware atom at run-time, enabling on-line compilation.

an Intel FPGA Cyclone V SoC device. Figure 6.13 is the floorplan for the initial FPGA test scheme. The FPGA is configured with two small reconfigurable islands capable of containing a hardware mathematical accelerator. Connections to the reconfigurable regions are controlled by the embedded ARM core; Figure 6.14 shows an expansion on the system, the regions are increased in both size and number. The total number of regions that the FPGA can be divided into is dependent on the total number of resources within the FPGA, less the required resources for the static logic, such as the reconfiguration manager, context switch controller and interfaces.

The processor can take an input file written in a HLL and use the reconfigurable fabric to construct accelerators from 'functional blocks'. This introduces a caveat, namely that each functional block needs to be pre-synthesised for each region of the FPGA. If there are a functional blocks and b regions on the FPGA and each block can go in any region, there are $a \times b$ partial configurations for the device. In some cases, for example floating-point mathematical functions, as in Chapters 3 and 4, the number of configurations for each region can be large. The requires large computational overhead to generate all configurations before they can be used. A large amount of memory is required to store all the configurations. All the partial configuration files are generated off-line and do not add to the computational load on the embedded processor.

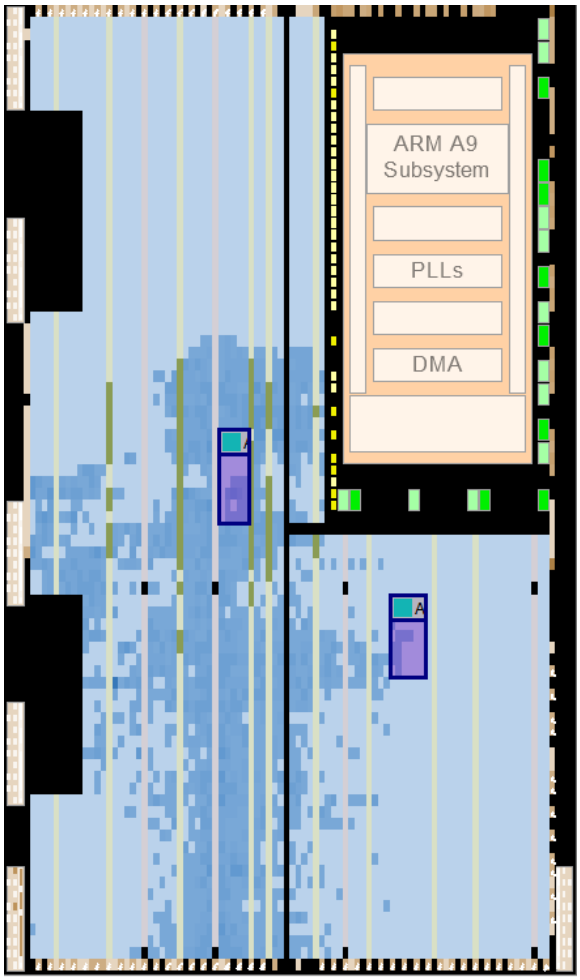


Figure 6.13: Intel FPGA floor plan view with two partial reconfigurable regions

The location of each functional block must be determined on a case-by-case basis at runtime. Chapter 7 will present a HLL synthesis tool that synthesises GLSL code into HDL. The tool constructs flow graphs where each node is a floating-point hardware accelerator. The on-line compilation method presented here uses similar techniques to extract control and data flow graphs from the source code. Each atom - base hardware unit - is then located based on constraints from the FPGA floor plan, shown in Figure 6.12. Additional constraints are added if FPGA regions are in use: the accelerator must be placed around the used blocks. This 'place and route' routine is coarser than traditional FPGA place and route. This processes adds some overhead to the on-line compilation, but it is far smaller than the time required for conventional FPGA synthesis. FPGA reconfiguration time adds further delay to the on-line compilation and configuration.

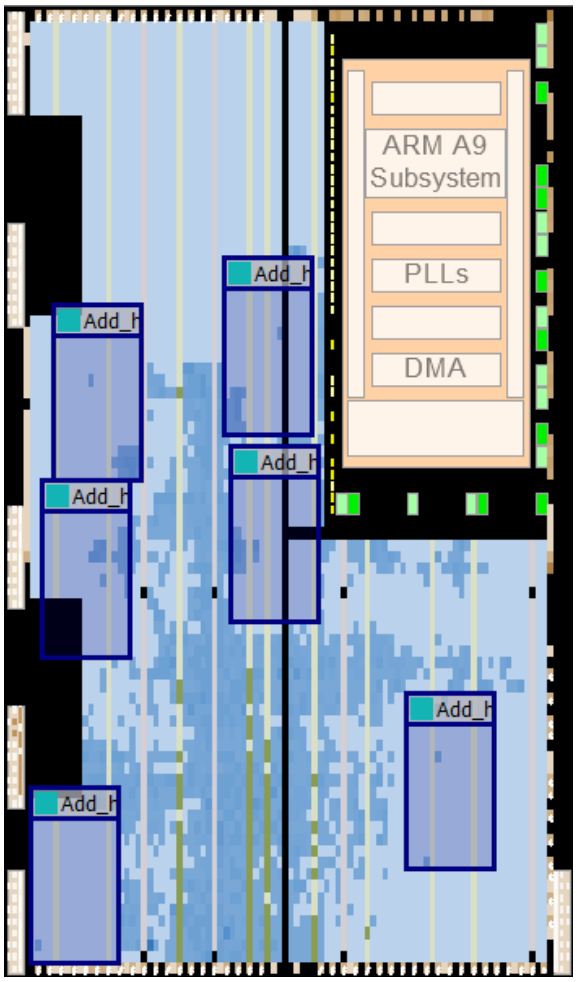


Figure 6.14: Intel FPGA floor plan view with six partial reconfigurable regions

6.3.1 Mapping to the FPGA's floor plan

Using methods similar to those that will be proposed in Chapter 7, flow graphs detailing program execution are generated. These flow graphs are re-mapped to match constraints of the FPGA chequerboard arrangement shown in Figure 6.12. Each reconfigurable region has a set number of interfaces. The local interfaces allow the logic to pass information only to its four neighbours. Hence a node can only influence, or be influenced by, a maximum of four other nodes. Global interfaces allow any region to talk to the HPS. The on-line compilation tool identifies nodes that are unable to support or be supported by the proposed chequerboard arrangement. Clones of unsupportable logic are made. The number of nodes to be cloned is determined by the number of nodes driven by parent nodes.

Figure 6.15 shows an example flow graph. It has already been identified that operations O3, O4, O5 and O6 can be fed by O2 and their respective variable V4 to V7. O2 is connected to five other nodes. It is not possible to connect one node with five other nodes, hence a clone must be made. The parent node of O2 (O1) only interacts with one node (O2) (V1 and V2

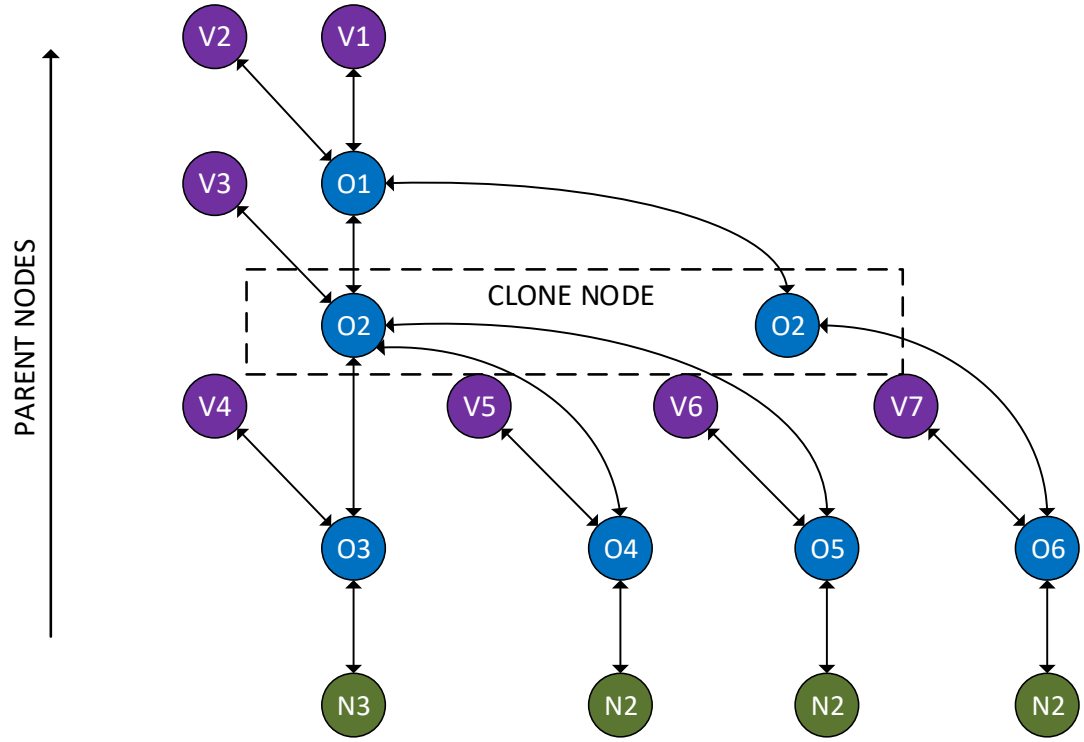


Figure 6.15: The control flow graph has been optimised to remove duplicated logic and save resources. The graph is rearranged based on the requirements of the chequerboard arrangement of the FPGA. Clones of nodes are made until each node is influenced by four or fewer nodes.

are found on the global bus). Therefore O1 has spare connections available so O2 is cloned and the clone is connected to O1. The children nodes O3 to O6 are distributed, as pictured. All nodes in the system now conform to the constraints imposed by the system.

The same logic can be applied even when V1 to V7 cannot be on the global bus. Including these additional connections to the system requires the nodes being driven by O2 and the clone of O2 to split differently. Node O5 is now moved to be connected to the clone of O2.

Once all the nodes satisfy the influence conditions, a suitable arrangement of the hardware is found. A flow chart of the algorithm used can be found in Figure 6.16. Each branch of the flow graph has a number of parent and children nodes, Figure 6.15. Each node on the graph contains its placement co-ordinates (default is $(0, 0)$). The placement algorithm starts with the top node of the first branch. A raster scan - scan through each horizontal position for a given vertical position, at the end of the horizontal, increment the vertical position and restart the horizontal scan - finds the first free node in the grid. At each empty square the number of adjacent free nodes is counted. If this number is equal to or exceeds the number of nodes influenced by the node to be placed, the node is placed. The co-ordinates in the

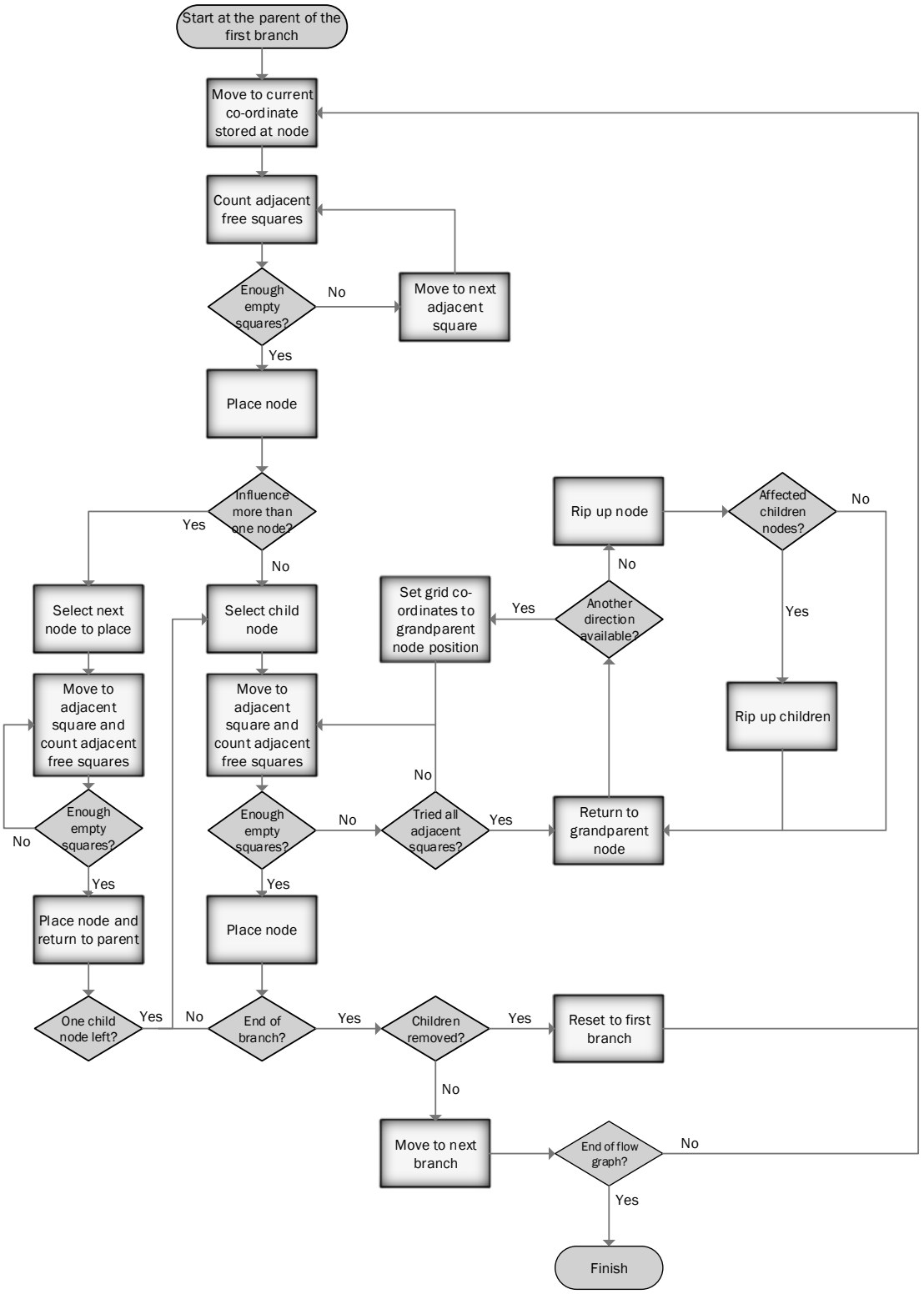


Figure 6.16: The algorithm used to map the control flow graph to the FPGA's floor plan. If the program reaches an error position, it is reported to the host and exits gracefully.

structure are updated. A *boolean* grid is used to represent the FPGA's floor plan, with a *false* entry denoting an empty square and a *true* entry a full square.

Once the parent node is placed all the children of that parent are also be placed. Each

child is systematically chosen and the squares adjacent to the parent are analysed to ensure they match the requirements of the child. When only one child is left, the process is repeated using the child as a starting point.

It is unlikely that this approach will result in a successful first-pass fitting. When a dead-end is reached the placements are back-tracked, and nodes are removed. If a node with more than one available direction for children is encountered the new direction is tested. If the new direction is suitable, nodes are relaid using the new path. If the top of the branch is reached, the next free space in the grid is chosen and placement restarts.

If nodes that influence more than one node are removed, all children are also removed. In situations where this occurs, the next pass will reset to the start of the flow structure to ensure any removed children may be replaced.

The system continues until all nodes are located. Once completed, each node holds a unique co-ordinate for instantiation in the FPGA. The associated files are passed to the reconfiguration controller to be loaded. Therefore, compilation of high level languages into FPGA logic at runtime is achieved.

6.4 Summary

This Chapter has discussed a number of considerations for the use of FPGAs as flexible hardware accelerators in heterogeneous systems. Dynamic reconfiguration provides the user with the ability to change the configuration of an FPGA at runtime. There are two types of dynamic reconfigurability: full and partial. Full reconfiguration completely scrubs the FPGAs original configuration and replaces it with a new configuration. Partial reconfiguration allows sections of the device to be reconfigured while the rest remains active.

Dynamic reconfiguration allows an FPGA to provide accelerators that would otherwise not fit on the device at the same time. A number of potential architectures for hardware accelerators implemented on FPGAs have been proposed, evaluated and discussed. Some architectures provide continuous end-to-end functionality, while others compartmentalise the FPGA for individual accelerators.

Novel context switching techniques for hardware have been presented. These allow defragmentation, or reassigning the FPGA's accelerators. Context switching allows the old tasks to be reinstated later, as though there was no interruption. Context switching was achieved by replacing D-type flip-flops with a proposed pre-emptible flop-flop. A hardware based manager was implemented to reduce save and load times. Discussions of methods to

context switch IP blocks were presented.

On-line compilation of hardware designs from a HLL input has been presented. The methodology allows an embedded processor to be passed a design file and generate the required hardware at runtime. The use of pre-synthesised atoms reduces overhead. Currently, synthesis tools can take hours to complete but the proposed method allows compilation to be mimicked at a higher level, reducing compilation time. The FPGA is divided into regions of reconfigurable logic with common interfaces to allow any configuration to be loaded, using dynamic reconfiguration, and communicated with its neighbours. However, each functional block must be pre-synthesised and loadable into each region. Input files are converted into flow graphs, as will be presented in Chapter 7. The flow graphs are re-mapped for the limitations of the chequerboard architecture. Although this still has computational overhead, it is significantly reduced from traditional HDL synthesis.

There are a number of limitations to be overcome. Data must be passed between clock domains and FPGA reconfiguration times are long. This technology is still in its infancy and reconfiguration times are likely to reduce with development. There is a vast amount of investment in this field, especially given the buy-out of a major FPGA manufacturer (Altera) by Intel in 2015.

The next Chapter will discuss the high level synthesis of OpenGL Shading Language (GLSL). The high level synthesis tool applies optimisations and produces HDL for inclusion in hardware design. The tool removes the need for a developer to be specialised in both hardware and software design.

Chapter 7

Automatic Synthesis of Hardware from High-Level Languages

Chapters 3 and 4 presented a number of floating-point accelerators. The efficiency and throughput of FPGAs makes them ideal as hardware accelerators. Chapter 5 used the floating-point accelerators to implement a Graphics Processing Unit (GPU) on a Field Programmable Gate Array (FPGA). From the GPU implementation it is clear there are challenges in developing custom hardware. Some approaches synthesise programmable Single Instruction, Multiple Data (SIMD) architectures to provide a more user friendly platform for programming [175]. This Chapter will present a High Level Synthesis (HLS) tool that accepts OpenGL Shading Language (GLSL) and produces the equivalent Hardware Description Language (HDL). The presented tool optimises the output HDL using critical path analysis and task scheduling.

To develop routines and procedures for a processor, a developer needs to only understand the procedural language. The compiler will interpret the procedural code and efficiently map it to the target architecture. The abstraction helps the designer and streamlines the design flow. Conversely, when designing hardware, the developer needs to both know the desired functionality and understand how to create the architecture. These are very different disciplines. Therefore, the barrier to entry for this technology is high.

There are a number of tools that are available designed to reduce the barrier to entry. A review of the performance of a number of these tools has been performed in [176]. These tools include: Bambu [121], LegUp [177], Vivado HLS [178], Intel (formally Altera) OpenCL compiler [179], Handel-C [180], Impulse-C [181] and Catapult-C [182]. Commercial tools are designed to use ANSI-C, or OpenCL-C in the case of the Altera OpenCL compiler. As the

tools abstract further from the target device and application, it becomes harder to produce an ideal implementation. Code optimisation is made harder due to there being multiple ways to express a function that give the same result. This can result in sub-optimised code.

Some of these tools are classed as real HLS tools and some are compilers. High level synthesis tools interpret the description of desired hardware behaviour, often given in algorithmic form, and produce the necessary hardware. Compilers take the description of hardware and convert this into primitives found on a target device. An extensive survey into various HLS tools and compilers is presented by Cordoso *et. al.* where a number of key features of the various tools are highlighted, [183]. The tool presented by this Chapter sits somewhere between the two tool types by interpreting the description of the graphics language and compiling this down for the target architecture.

A comparison of HLS tools [176] concludes that "no single tool produced the best results for all benchmarks" and "optimisations that are necessary to realise high performance in hardware differ significantly from software-oriented ones". Different tools apply different optimisations. There is no 'correct' way for the tools to perform operations. However, for a given application one tool may work better than another.

7.1 High level versus low level

High-level and low-level describe the amount of abstraction offered by a programming language. Very high-level languages, such as Java, abstract the user away from the platform to such a level that any application can run on any platform. Applications are not pre-compiled for a target device. Languages such as C still offer a high level of abstraction, but are subject to some architectural dependencies.

Assembly language is very low-level. There is no abstraction from the target architecture, and designs must use the instruction set for the target device. Unless two devices have the same instruction set, it is not possible to move code between devices. Modern processors tend to share a minimal working instruction set, helping to increase the portability of lower level languages. In general, the lower the level of a language, the more optimised the result (assuming the developer knows what they are doing).

7.2 Traditional design flows and optimisation techniques

Traditional compilation tools use languages designed to run on a specific platform and compile them down for the target architecture. For example, gcc and g++ target General Purpose Processors (GPPs) and Spectra-Q [184] targets FPGA architectures. The language syntax and specific compilation chain help to make optimisation easier.

There are a number of areas that lend themselves to optimisations: loops, recursion, and memory. Loops and recursion can form bottlenecks. Common techniques for evaluating loops use unrolling to express the complete contents of a loop (gcc compiler). Memory accesses pose a different optimisation problem. Limiting the number of memory accesses decreases the execution time of a procedure.

When synthesising custom hardware from High-Level Languages (HLLs), similar optimisations must be performed. Unrolling loops in hardware leads to large resource cost, as shown in the performance optimised implementations in Chapter 3. Chapter 3 presents an alternative method to implement a loop that reduces resource cost but increases latency. Loops with a large number of operations will still incur a high resource cost. Methods for performing behavioural synthesis from HLLs that include recursion and looping are discussed in [185].

Recursion can be considered to be similar to loops. A number of tasks are called in sequence, which can result in the size of the program growing. When tasks are called recursively, data can be placed in memory as the processor moves through the recursive calls. Modern processors have a reasonable amount of memory, making recursion possible, despite the potential to cause memory leaks and access violations. In a hardware environment, every single resource has to be accounted for. Recursion can use significant numbers of resources. Synthesising an FPGA equivalent of a heap with automatically generated interfaces is presented in [186]. The more a hardware implementation looks like a processor, the greater the abstraction, and the smaller the performance gained from using hardware.

7.3 High-level synthesis of OpenGL shading language

A HLS tool converts code written in a procedural language for implementation in hardware. The use of commercial HLS tools still requires an appreciation for the different architecture.

Chapter 5 presented a design for a GPU on an FPGA. The performance metrics demonstrated the FPGA performing faster or more efficiently than similar, embedded GPUs. GPUs

use shading language (GLSL, HLSL). Designing equivalent hardware for a GLSL shader presents challenges. Using a HLS tool mitigates design challenges for the designer, producing hardware shaders for reconfigurable architectures.

GPUs and FPGAs are parallel devices, therefore the FPGA lends itself to performing graphics processing. The FPGA is also a highly versatile device. Due to the FPGA's reconfigurability, it can be used to accelerate a range of tasks, as discussed in Chapter 6. To get the greatest performance benefit, the FPGA's architecture is reconfigured to match the task. Similarly, a GPU can be used to perform non-graphics tasks. However, the task must be reformed to match the GPU's architecture, losing performance.

A compiler starts by extracting data and control flow for the input file. The flow is represented as a graph. Representing the information as a flow graph makes performing optimisations easier. Figure 7.1 shows the algorithm for parsing the input GLSL file and building the initial flow graph structure.

Figure 7.2 shows the construction of the flow graph before optimisations. All optimisations are performed on the flow graph. In the example flow graph, Figure 7.2, three types of operation are shown. Node 1 (N1) is an assignment operation where Variable 1 (V1) is copied to another variable. Node 2 (N2) is a single step mathematical operation, such as $a = b \times c$. Operation node O1 contains the operation to be applied to the two inputs, V2 and V3. The result is placed in the output node, N2. Node 3 (N3) represents how the graph stores a series of consecutive operations. The top of the tree consists of two variables, V6 and V7, which are subject to the operation in O4. The result from O4 is fed to the next operation, O3, along with V5. This is repeated until the answer is formed and passed to the root node, N3. Operations that feed new operations have a temporary register assigned, Figure 7.3.

Figure 7.4 demonstrates how the flow graph represents data flow - values - and control flow - operations. The graph models the functionality described by the GLSL file that is to be mapped to reconfigurable hardware. Performing the synthesis at this point would result in a potentially inefficient solution.

7.4 Optimising the flow graphs before synthesis of hardware

The tool applies a series of optimisations to the graph before writing out the HDL. Figure 7.5 shows the optimisation algorithm for removing redundant hardware and for performing critical path analysis.

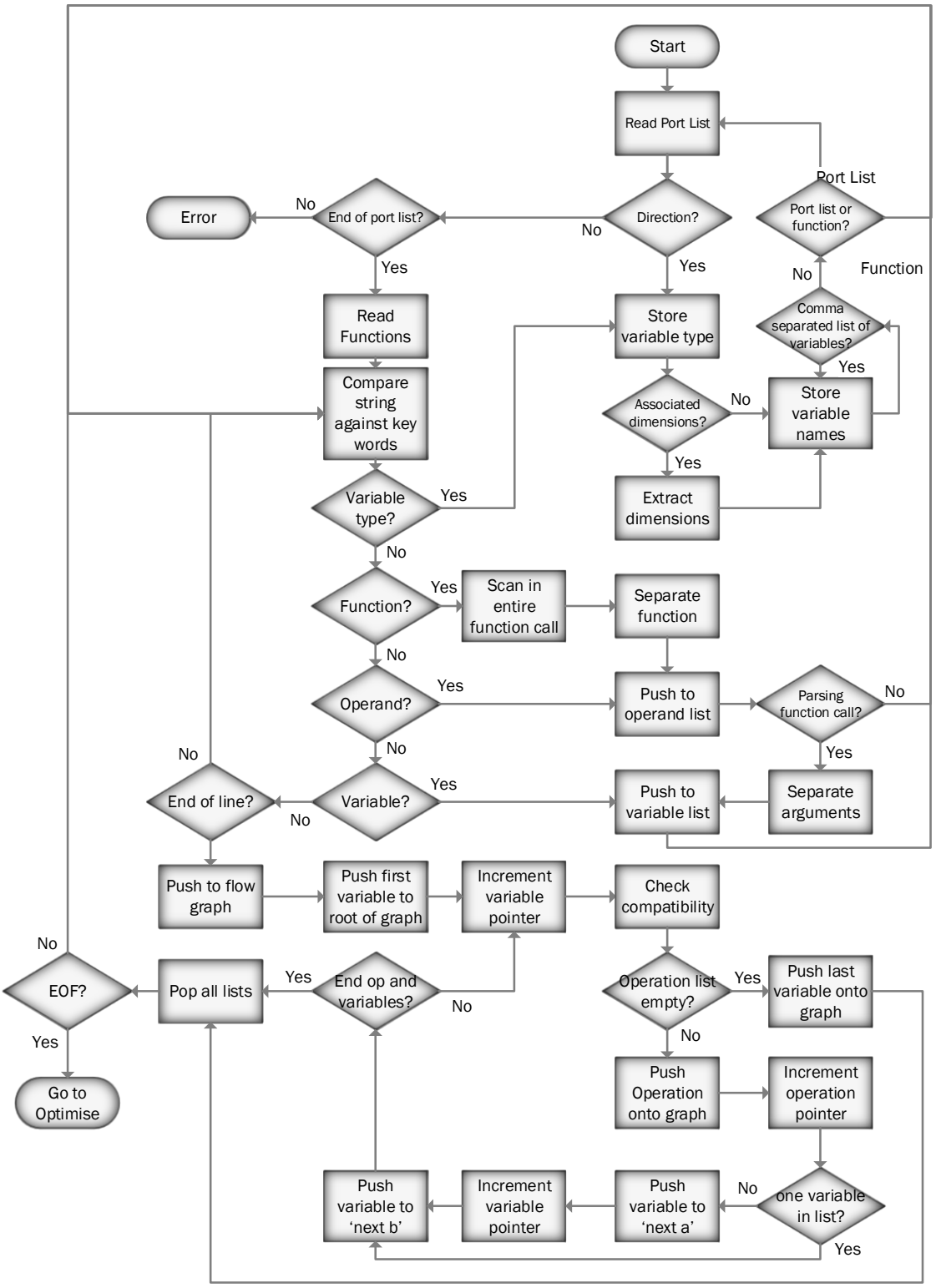


Figure 7.1: Flow diagram depicting the parsing and interpretation of the GLSL file to construct the data flow graphs.

GPU's have a high throughput, therefore it is important that the FPGA retains a high throughput. The flow graph is analysed for operations that can be implemented in parallel. Despite the need for high throughput, resources are limited and care must be taken. It

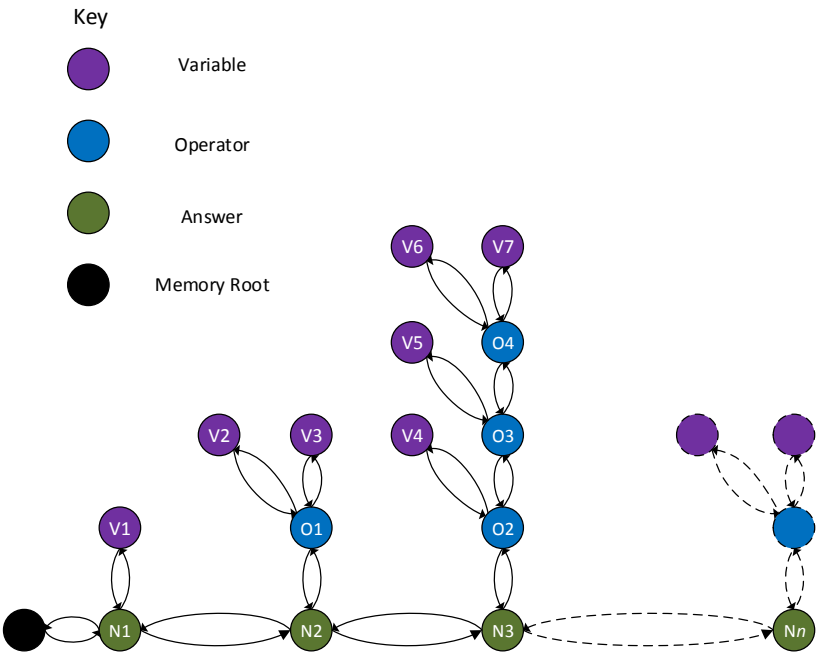


Figure 7.2: Example flow graph created by the initial parse of the input files. The graph has root nodes that are linked to a tree containing information about the variables and operations required to generate the root.

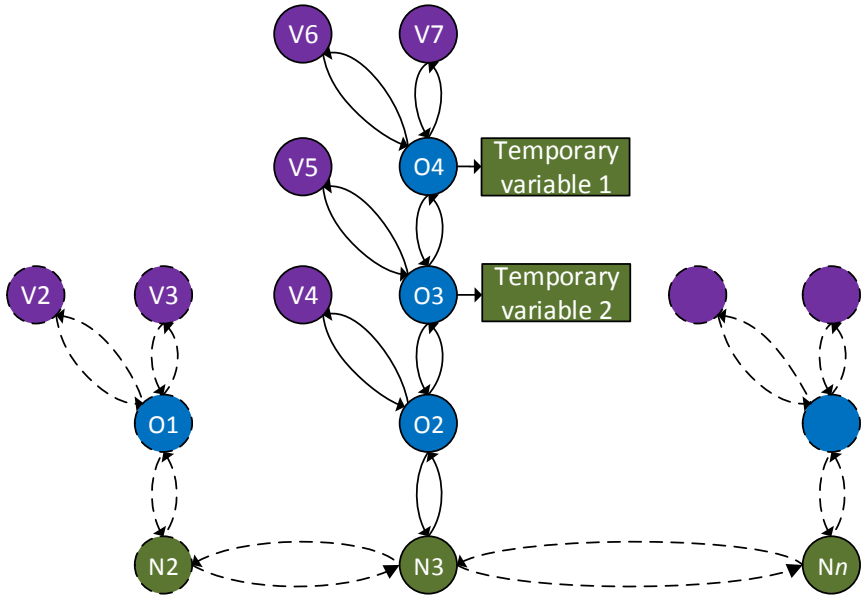


Figure 7.3: Variables that require more than one operation to create will have temporary variables assigned to each operation. When this is translated to hardware it allows construction of the GLSL code using a library of pre-defined blocks.

is possible to implement dynamic memory in hardware [186], however, this can reduce the performance of the system. It is vital to identify every location where resources can be saved. This includes delay chains and removal of repeated functions. Critical path analysis identifies

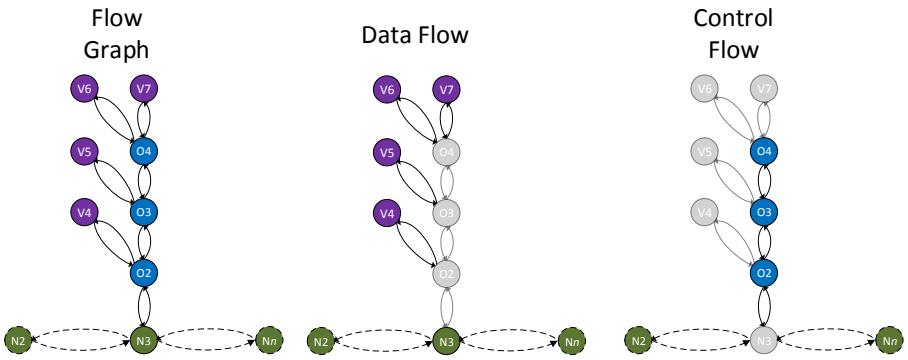


Figure 7.4: From the initial flow graph the data and control flow of the input file can be extracted.

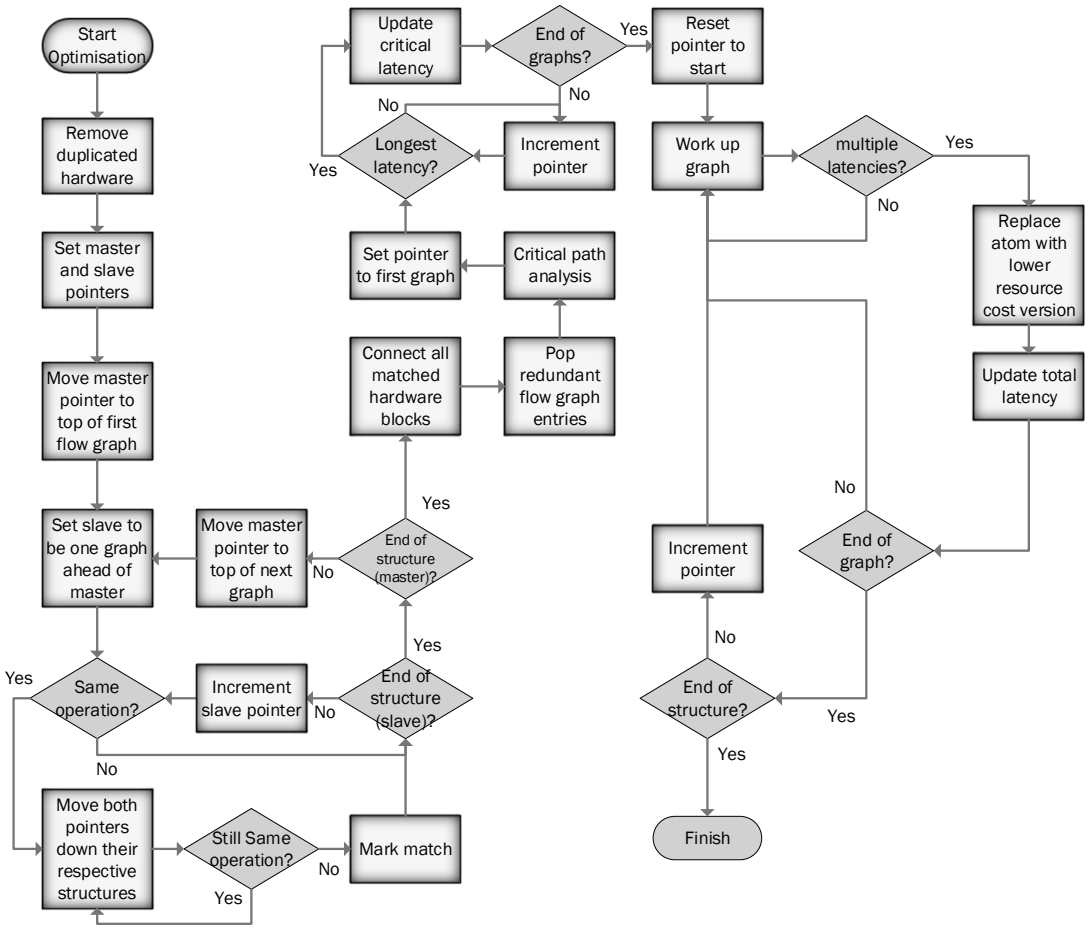


Figure 7.5: The two key optimisations performed are used to remove duplicated hardware and to schedule tasks based on critical path analysis. This flow diagram gives an overview of their operation.

which nodes can be replaced with more resource efficient variations without increasing the overall latency.

The synthesis tool uses 'base' units called atoms. The atoms represent the hardware's instruction set. In this case study the atoms are the hardware accelerated functions discussed

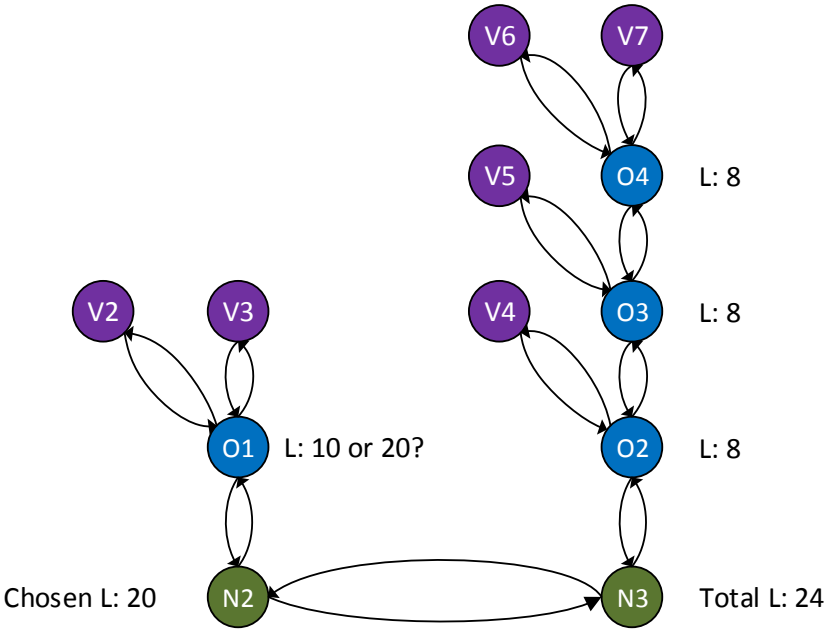


Figure 7.6: The synthesis engine has a set of predefined atoms that are used to build the HDL from. Each of these atoms has a known latency. By parsing the control flow structure, the total latency of each chain and hence the critical path is found. This information is used to select which version of an atom to use that keeps the overall resource count minimal.

in Chapters 3 and 4. The floating-point maths functions are independent of a particular vendors architecture, meaning the synthesis tool can be included as the front end of any synthesis chain. The atoms have been especially designed to work with this tool. When the tool is run, a custom data path is constructed that is tailored to the graphics shader being implemented. A number of optimisations are performed to reduce the resource overhead, such as identifying where lower cost atoms may be used for functions not on the critical data path.

7.4.1 Critical path analysis

Each atom has an associated latency value. These values are known by the synthesis tool and are used for critical path analysis. The critical path analysis is used to make an informed decision about task scheduling for functions not in the critical path.

Chapter 3 demonstrated methods for optimising hardware implementations for either resource or performance. In the example in Chapter 3, a resource optimised implementation used as few as one adder and one multiplier. It is possible to increase the number of adders and multipliers, reducing the latency. This can be done until enough multipliers and adders are added to create the performance optimised implementation. Each implementation has a

different resource cost and latency. Figure 7.6 shows the critical path of a flow graph to be the path with the highest latency; latency values are given in clock cycles and the issue rate (iteration interval) of implementations is one unless otherwise specified by the tool. In this path, all implementations will use the lowest latency versions of nodes if there is a choice. Elements on other paths may also have a choice of implementations. On the non-critical paths, the chosen implementation has the the highest latency (and therefore lowest resource count) so long as the overall path latency does not exceed the critical path. This scheduling operation makes the best use of resources while keeping the overall execution time as low as possible.

7.4.2 Removing repeated hardware

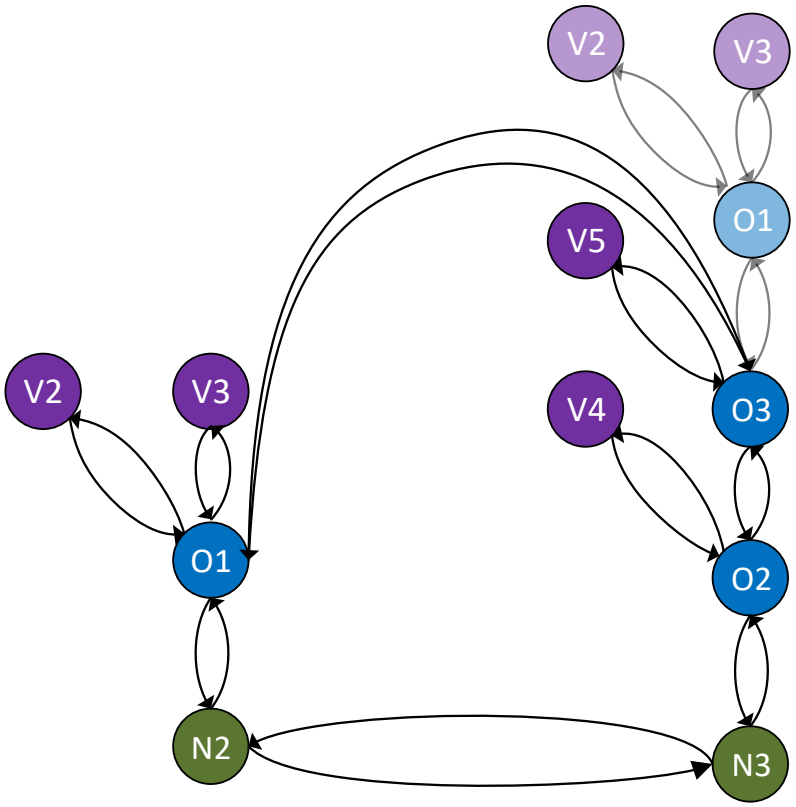


Figure 7.7: Analysis of the flow graphs is used to determine repeated structures. The graph is rearranged to re-use hardware and fan the output out to multiple locations the overall resource use is reduced. The shaded out nodes (V2, V3, and O1) have been removed to save on resources.

Resources in hardware are limited. Modern FPGAs are significantly larger than their predecessors, however, every resource still counts. Implementing several sets of identical logic that process the same information and create the same result is redundant. It is not

uncommon for a program to calculate a variable for use in multiple locations. This is identified and marked to ensure only one set of logic is used in the final implementation.

Repeated operations can be created in more than one way. Operations can create temporary variables that are used later in the code. Otherwise compound operations may repeat the same operations on the same variables. If this is the case the flow graph has a set of repeated nodes. The flow graph is parsed and comparisons are made between all operations and their associated data. If a match is discovered this is marked so that the tool knows to only implement one set of logic and fan-out the output to many locations, shown in Figure 7.7.

7.5 Synchronising the data path

Processing the inputs to the outputs takes time. Not all data paths are the same length. On paths that are not on the critical path, modules are selected to have the highest latency possible. This will give the most resource optimised final implementation. There is no guarantee the path latency will be the same for every path. The final structure is analysed for mismatching latency and delay chains are used for re-synchronisation. Synchronising the data creates pipelines where data is moved on every clock cycle. The timing and synchronisation for the FPGA is less complicated and the overall performance is improved. The required delay for each path is stored in a control flow graph, shown in Figure 7.8.

Input data and generated data may also need to be delayed. If incoming data is not used right away, accepting new data on the next clock edge causes the previous data to be lost. When the original data is required the new data would be used instead, resulting in invalid or unexpected results. Figure 7.2 shows some variables are not needed until previous operations have been completed. Similarly, some operations use variables created by other operations, but the preceding operations may have latency mis-match. Implementing a delay structure propagates the original data through a set of registers until it is required by the function. The length of this set of registers is determined by the latency of the operations that are performed before the data is required.

Some data is required in more than one location at different times. Figure 7.9a presents a method for delaying data by different numbers of clock cycles. Each delay has its own delay line. Although functional this is an inefficient way to use resources due to duplication. The control flow graph for delay, Figure 7.8, groups matching variables together. Matching variables are sorted in ascending delay length. New entries are appended to the structure with their required delay length, therefore creating the delay control graph. Any variables

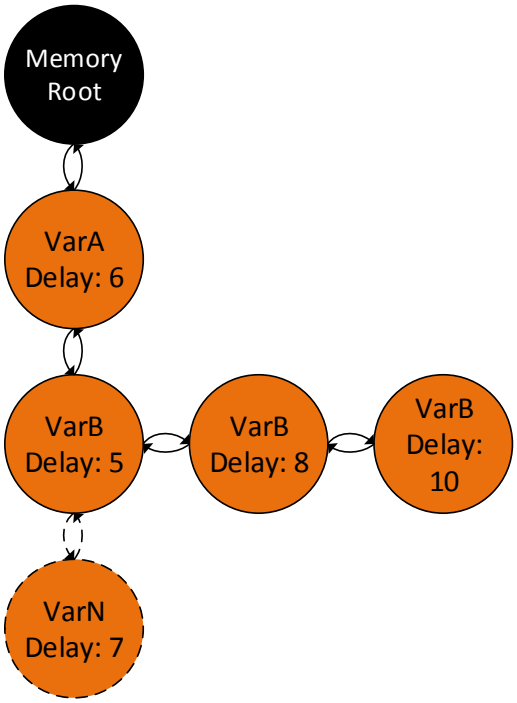


Figure 7.8: Ensuring the module is synchronous is key to pipelining. It is unlikely that each data path through the module has the same latency. Analysis of the control flow graph gives the critical path. From this any additional latency that needs to be applied to a variable can be determined and stored in another control flow graph.

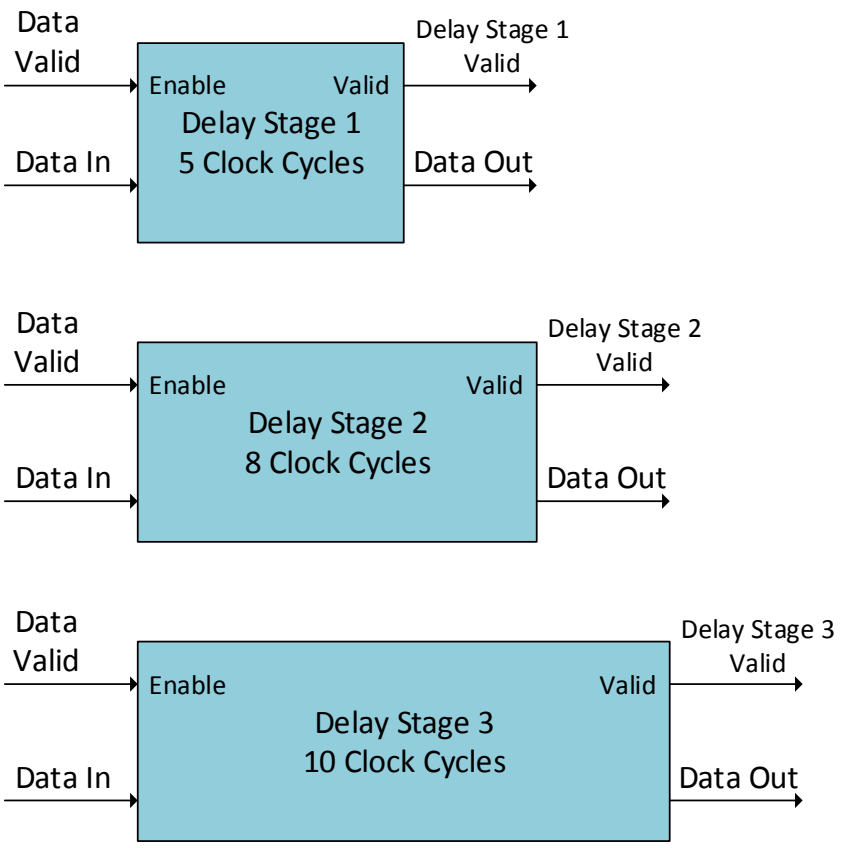
that have more than one associated delay length have the difference in each delay calculated. The delay structure is created as a single delay pipeline with multiple tap points for all the stages where the variable is required, shown in Figure 7.9b.

Similar techniques are used to store delay values for the output. Variables will only ever need a single delay length. Figure 7.10 shows the insertion of delays for the output stage.

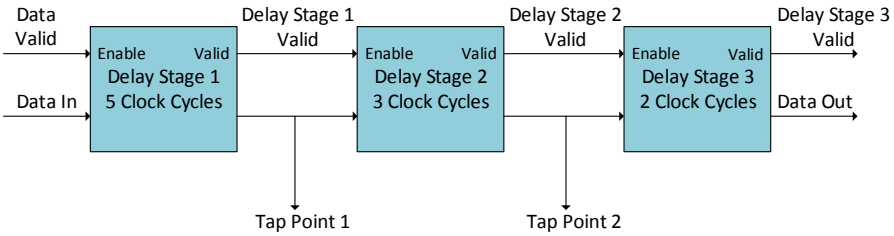
7.6 Pipelines and resource reuse

The previous two Sections (7.4 and 7.5) discussed both saving resources and using more resources. Pipelining allows faster switching of the FPGA fabric and therefore higher overall performance but increases resource use. However, resource reuse is required to limit the size of a design. Chapter 3 demonstrated the use of feedback to create a resource optimised matrix-vector multiplier. In order for this to work the atom cannot accept new data on every clock edge while it is busy. This violates the pipeline requirement. Similarly, dividers and multipliers sometimes have early exit conditions.

Creating implementations that do not reuse logic is not always practical. Inexpensive devices, such as the Intel Cyclone range, have limited resources and it has already been



(a) Duplicated delay chains of varying lengths.



(b) Single delay chain with multiple tap points.

Figure 7.9: Managing the amount of resources used when pipelining signals is important. Implementing multiple delay chains that duplicate each other leads to a resource inefficient implementation, 7.9a. Instead using a single delay chain with tap points at the required lengths uses fewer resources, 7.9b.

shown that graphics rendering designs can grow in size very rapidly (Chapter 5). Further, using performance optimised implementations on the non-critical paths wastes resources and increases power consumption with no performance gain. In order to maintain an effective pipeline while allowing resource reuse, a 'wait' signal is generated. The wait signal is back propagated from modules that reuse resources to all parent modules, shown in Figure 7.11. This was discussed in Chapter 5.

Information for nodes that require a wait signal is stored by the HLS tool presented here.

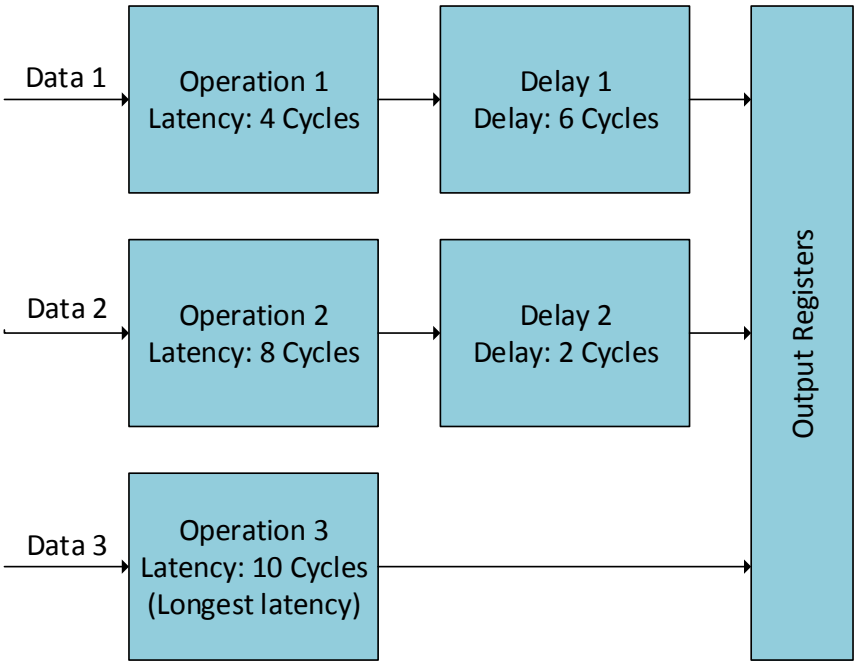


Figure 7.10: To maintain synchronisation through out the graphics processor, the outputs for each module are delayed such that they all occur on the same clock edge.

Latency values also take into account the effect of the wait signal. The synthesis tool uses stored information that details if a module requires a wait signal, generates it and connects the appropriate busses.

7.7 Using the automated synthesis tool

The synthesis tool is designed to be used in the same way as a GLSL compiler would be used for a target GPU. It works from an input GLSL file. The tool can be used at the start of hardware synthesis and the output is a binary file that can be loaded straight into an FPGA. This allows the replacement of GPU-SoC devices with FPGA-SoC devices. Chapter 6 discussed the use of dynamic reconfiguration to change FPGA configurations at runtime. The binaries created with this tool can be used for dynamic reconfiguration.

For the purposes of testing the tool, the types of functions used in GLSL are categorised by data type and format. GLSL works primarily using vector and matrix data types, along with the conventional types such as boolean, signed and unsigned integers, and single- and double-precision floating-point. Operations are in: *variable operand variable*, or *ReturnType function(argument list)*, or a combination thereof. A number of tests were performed where the input GLSL files covered functions in these formats, Table 7.1. The output HDL was synthesised, simulated and tested.

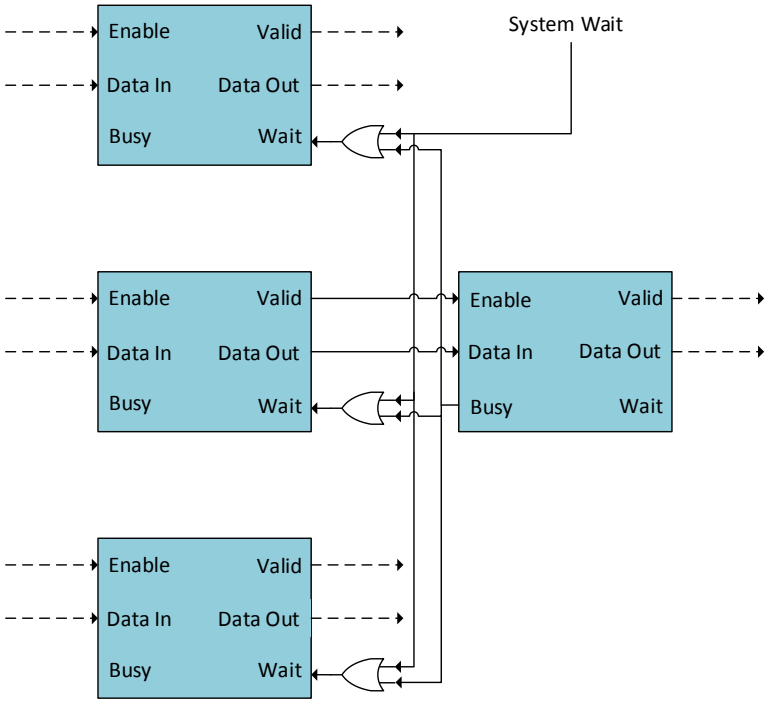


Figure 7.11: Some modules re-use resources due to size constraints. In this situation they must prevent data being loaded into them before they are ready. This is achieved by generating a 'wait' signal that is propagated backwards through the system to pause operation of earlier modules.

Table 7.1: GLSL shaders can range from straight-forward operations such as moving data from the input to the output side for use in a later shader, to complicated matrix and vector operations. A number of increasingly complicated GLSL operations have been implemented as GLSL shaders that have been converted by the HLS tool.

Shader	**Registers**	**ALMs**	f_{max} **MHz**
Assignment	2	2	580.04
Floating-point multiplication	299	135	244.32
Vector-Matrix multiplication	11,556	4,808	155.62
Matrix-Matrix multiplication	143,298	61,455	135.85
Vector-Matrix-Matrix multiplication	154,420	66,377	139.45
Square-root	473	205	276.63
Exponential	76	48	82.17
Vector length	55,166	16,224	209.42
Vector normalisation	62,238	18,785	220.17
Vector dot product	11,667	3,555	349.04

Chapter 5 presented, tested and implemented a number of shaders. These have been used to further demonstrate the functionality of the synthesis tool. Metrics are given in Table 7.2, which also gives the throughput of the same operations performed on embedded platforms. The matrix-matrix and vector-matrix-matrix operations have a high resource count as they are implemented as fully unrolled pipelined designs with an issue rate of one, as per Chapter 3. It can be seen that the throughput of the hardware synthesised by the tool is comparable to hardware designed in Chapter 5.

The generated hardware has a higher resource cost than the hand-designed hardware. This is attributed to the rules regarding synchroniser chains that must be followed by the synthesis tool. If a set of data is presented on the input of the module and required on the output, such as an RGB colour value, the synthesis tool creates a delay chain to ensure this appears in time with the rest of the outputs. In the case of the modules designed in Chapter 5, the design considered the whole system. When hand designing, a decision was made to not delay the RGB value for static, constant shading.

All the examples presented here function as expected under simulation and real-world implementation.

7.8 Summary

In this Chapter the high-level synthesis of hardware was discussed. The design for a HLS tool for synthesising hardware accelerators from GLSL was presented. The tool is designed to reduce the barrier to entry for FPGA technology in heterogeneous environments.

There are a number of research and commercial available tools that exist which allow ANSI-C or OpenCL-C to be used with reconfigurable hardware. However, the more abstract a tool becomes, the harder it can be to get the 'perfect' solution from it. HLS tools can perform a number of optimisations but this does not always provide the best result for all situations.

The tool presented in this Chapter uses a different input language to the commercially tools, with a specific target application. It is believed this is the first example of its type. The tool uses a number of specially designed atoms. The atoms are optimised for the purpose of the hardware acceleration of floating-point mathematical functions.

The synthesis tool produces and optimises flow graphs that represent the input GLSL file. Optimisations include critical path analysis, task scheduling, data pipelining and identifying areas where resource reuse can be implemented. Critical path analysis informs decisions

Table 7.2: Automatic generation of vertex shaders are compared to 'hand-written' variants for the same function. Three different resource/performance optimisations were chosen to compare FGPA resources and throughput. Included are some throughput figures for other embedded technologies.

Shader	**Registers**	**ALMs**	f_{max} **MHz**	**Vertices per second**
Hand-Written (Performance)	17,624	7,485	154.15	154.15M
Automatically Generated (Performance)	20,236	8,266	151.08	151.08M
Hand-Written (Hybrid)	5,756	2,524	150.29	37.57M
Automatically Generated (Hybrid)	8,368	3,196	158.58	39.65M
Hand-Written (Resource)	3,177	1,474	161.71	8.09M
Automatically Generated (Resource)	5,789	2,187	163.51	8.180M
Cyclone V embedded ARM Cortex-A9 Software Rendering	-	-	-	170k
Allwinner A13 device comprising Cortex-A8 CPU with Mali 400 GPU	-	-	-	35M
Allwinner A13 device using Cortex-A8 Software Rendering	-	-	-	32.73k

about delay chains to ensure data remains pipelined and synchronised. The output of the tool can be included directly into the FPGA-GPU design presented in Chapter 5. Further, the tool does not require the designer to have any specific hardware knowledge.

The next Chapter will present the conclusions from the topics covered by this research. The benefits and limitations of FPGA based heterogeneous systems are summarised, and future works are presented.

Chapter 8

Conclusions and Further Work

This research has covered a number of topics. Chapter 3 demonstrated implementations for a number of basic floating-point accelerators in hardware. Two case studies for using these accelerators were given. The operations in Chapter 3 are easily implemented in FPGA technology. The implementations have a maximum relative error of one Unit of Least Precision (ULP), which complies with IEEE-754R. For almost all the functions, with the exception of floating-point addition, only a few FPGA resources are required, even at double-precision. Floating-point addition is known to be a costly operation compared to fixed-point addition. All operations have a high throughput, in excess of 100 MFLOPS at double-precision.

Chapter 4 presented hardware implementations of more complicated mathematical functions that are more challenging to implement. Implementing iterative approximations is resource or time intensive. Two alternative approaches, non-recursive and curve-fitting algorithms, were presented. These have a much lower resource cost, while maintaining a high throughput. The non-restoring algorithm achieved a relative error of one or fewer ULPs. Curve-fitting methods exhibited much higher relative error, but maintained a low normalised error. Implementations that used curve-fitting techniques had a considerably lower resource use than Euler expansion or power series methods. Chapter 4 also presented an implementation of the Hodgkin-Huxley model of a neuron on the FPGA.

Chapter 5 created an graphics processor in hardware that was based on a standard OpenGL graphics pipeline. The FPGA-GPU consisted of a vertex shader, a rasterizer and a fragment shader. Each stage was created using the floating-point accelerators presented in Chapters 3 and 4. The throughput and efficiency of the FPGA implementation was compared with other embedded processors: an ARM Cortex-A8, an ARM Cortex-A9, an ARM Mali-400 GPU, an NVIDIA Tegra K1 and an NVIDIA GTX 780. The metrics demonstrated

that the throughput of the FPGA implementation was greater than the ARM devices and always more efficient all the embedded devices. Chapter 5 discussed methods for increasing the throughput of the design instantiating multiple modules in parallel to remove bottlenecks, and compared throughput and resource use between forward and deferred rendering techniques.

Chapter 6 detailed the use of full and partial dynamic reconfiguration of FPGAs. Dynamic reconfiguration allows the FPGA to change which hardware accelerator is implemented at runtime. In addition, Chapter 6 presented methods for context switching hardware accelerators and on-line compilation and synthesis of FPGA configurations. Context switching the hardware allows the accelerators in the FPGA to be de-fragmented or swapped out and returned to without loss of data. The on-line synthesis method requires a pre-defined FPGA arrangement and pre-synthesised nodes. The nodes are arranged by extracting and optimising program flow from an input file.

Chapter 7 discusses High Level Synthesis (HLS) of hardware. A number of commercial tools that all take C based languages (ANSI-C and OpenCL-C) were discussed. The design for a new tool that accepts OpenGL Shading Language (GLSL) has been presented. The presented tool applies optimisations to the flow graph representation of the GLSL file. Optimisations use critical path analysis, task scheduling and resource reuse identification to increase the performance and limit the resource cost of the output hardware.

This research has highlighted a number of benefits and limitations of using FPGA-SoC devices as flexible hardware accelerators in a heterogeneous architecture.

8.1 Benefits

Hardware acceleration provides higher throughput or more efficient performance compared to processor-based implementations. However, there is a trade-off between hardware flexibility and how often the hardware is used. Increasing the flexibility of the hardware accelerator reduces the performance gain.

This research asserted that a wide variety of applications can be created from combinations of mathematical operations. To demonstrate this a number of implementations of floating-point mathematical operations have been used to create two different applications: neuron modelling and graphics rendering.

The Hodgkin-Huxley model of a neuron uses the exponent in the transfer function. The hardware implementation of the exponent function had a high relative error due to the curve-

fitting technique. The case study demonstrated that a high relative error in certain applications does not prevent function. The trade-off between accuracy and resources allowed large, complicated systems to be implemented with a smaller number of resources while maintaining functionality.

The implementation of a Graphics Processing Unit (GPU) on an FPGA demonstrated a case where being able to change the hardware's functionality at runtime is beneficial. The performance of the FPGA implementation was compared against a number of commercial processors: ARM Cortex-A8 and A9, Mali-400 GPU, NVIDIA Tegra and NVIDIA GTX 780. A normalised metric of vertices processed per second per Watt showed the FPGA implementations, even configured for lowest resource cost, are more efficient than the commercial processors. It is shown that FPGAs provide a platform that is, at worst, comparable to similarly priced commercial devices.

Dynamic reconfiguration allows the configuration of the FPGA to be changed at runtime. In a heterogeneous environment, this allows tasks from the processor to be off-loaded to the FPGA for acceleration. The accelerators in the FPGA can be changed to match the current processor tasks, increasing the flexibility of the FPGA. Unlike using a GPU-SoC device to accelerate processor tasks, using an FPGA-SoC allows customisation of the hardware architecture to match the task being accelerated, maximising the performance gain from the FPGA.

8.2 Limitations

This research has highlighted a number of limitations yet to be overcome.

Chapter 4 detailed implementations of floating-point functions that require iteration to approximate the answer. Conforming with IEEE-754R by iteration in hardware results in a resource or time intensive design. A number of alternative methods were proposed that reduced the resource use. However, these did not always conform to a relative error of one or fewer ULPs. It was demonstrated through case studies that a high relative error did not prevent functionality of the application. In some applications, there are quantising factors that make the high relative error insignificant.

Reconfiguration times for FPGAs are still significant. Dynamic reconfiguration requires the reconfiguration file to be loaded to the reconfiguration controller. Moving the file to the reconfiguration controller is slow, even with Direct Memory Access (DMA), increasing the reconfiguration time. It is likely that future revisions from vendors of FPGA-SoC devices will

address the speed limitations of dynamic reconfiguration. For applications that are being run for a prolonged period, a reconfiguration time of a second is negligible.

Managing the hardware accelerators implemented in the FPGA is a considerable task. The most optimal use of the FPGA allows the accelerators to be changed based on the current processor's application. A single application could have a number of accelerators and multi-cored processors can run multiple applications. Chapter 6 presented methods to context switch and change the accelerators at runtime, and even compile new accelerator chains from an input file at runtime. However, the accelerators need to be ranked based on importance and which accelerators are currently implemented needs to be tracked. Additionally, the FPGA resources are limited. It is not always possible to implement all the most 'important' accelerators, although smaller, less important accelerators may still fit. Determining which accelerators to use adds complexity and computational overhead. FPGA-SoC devices with multiple processor cores could dedicate an entire core to managing the implemented accelerators and processing requests from the processor.

Moving data between the FPGA and the processor often requires crossing a clock domain. There are a number of techniques for safely crossing clock domains, for example synchroniser chains and dual-clock First-In, First-Out (FIFO) buffers. Unfortunately these add latency and reduce the effective performance gain of the system. Additionally, implementing accelerators that work on data that is frequently accessed by the processor increases overhead in maintaining cache coherence. Ideally, accelerators will work on data that is never, or infrequently, required by the processor. It must be ensured that any gain from using the FPGA for acceleration is not off-set from moving data around or maintaining cache coherence. This adds a further condition when identifying which routines are most beneficial to accelerate.

Designing custom hardware for an FPGA presents a high barrier to entry. Using FPGA technology requires the designer to specialise in both hardware and software development. HLS attempts to remove the barrier to entry by synthesising hardware from other languages, for example ANSI-C. Optimising designs from procedural code for implementation in hardware is a challenge. Commonly used procedural constructs, such as loops, recursion and memory access do not map to FPGA technology without considerable cost. Iterative functions, like loops and recursion, can unroll creating large hardware. The tool presented in Chapter 7 restricts the end application to greater allow optimisations to be applied to the hardware. The use of GLSL as an input language helps with mapping to FPGA technology, due to similarities between GPUs and FPGAs - both are parallel architectures. Further,

GLSL is designed to work with vector maths and does not support loop constructs.

8.3 Future work

8.3.1 Hardware accelerated functions

From this research a number of future works are identified. Chapters 3 and 4 present implementations for a set of mathematical operations. The operations that have been implemented do not constitute a full library of floating-point functions. The implemented functions should be expanded upon to cover all known mathematical functions. The performance and resource cost for all functions should also be analysed. For a number of functions, for example logarithms and trigonometric functions, alternative methods may need to be researched to decrease resource cost and increase throughput. Chapter 3 discussed optimising implementations for different properties, resource use and performance. It has been shown that a number of floating-point functions, such as matrix-vector operations, can be created using a repeated pattern of smaller functional units. Research into developing tools to automate this process based on a number of parameters, resource cost, issue rate and power consumption, could be conducted. A complete library would cover all possible optimisations for all implementations.

8.3.2 High level synthesis

The HLS tool presented in Chapter 7 uses the implementations from Chapters 3 and 4. Completing the library of implemented functions means the HLS tool support can be extended to cover the complete GLSL function list. Additionally, the library of floating-point functions will provide a number of latency options that the HLS tool will be able to use when performing critical path analysis and task scheduling. Further, investigation into HLS support for other languages and optimisation techniques could be conducted.

8.3.3 Dynamic reconfiguration of FPGAs

8.3.3.1 Context switching and hardware accelerator management

Chapter 6 presented a number of techniques for use in a dynamically reconfigurable environment. It has been identified that managing dynamically reconfigurable accelerators is complex. Further research will implement a full heterogeneous manager that determines which accelerators are the 'best' to implement given current processor tasks, and maximises performance gain from the FPGA. Context switching methods covered by Chapter 6 pre-

sented the pre-emptible flip-flop to replace conventional D-type flip-flops. Discussions were presented for extending this to context switching hard IP blocks. With the aid of FPGA vendors to remove the IP barrier, these methods will be further explored and developed. This research topic extends to exploring ways to interface the FPGA and processor to minimise overhead while moving data between the two.

8.3.3.2 On-line synthesis and reconfiguration time

Chapter 6 also discussed methods for on-line compilation and synthesis of hardware accelerator chains. The individual components of this have been implemented and shown working. Future research will combine all the individual aspects to present the continuous end-to-end on-line flow, using a next generation FPGA-SoC device. A variety of FPGA architectures can be developed that will allow coarse grain on-line reconfiguration. This forms the basis of finer grain on-line reconfiguration. This will require research into methods for decreasing expensive place and route algorithms. In addition, it has been identified that reconfiguration times for FPGA-SoC devices are a limiting factor. Methods to decrease reconfiguration time will be explored with the aid of FPGA vendors.

8.3.3.3 Thermal planning

It was noted in Chapter 2 that thermal planning is critical for ensuring the life time of the FPGA. The thermal effects of over-using the same areas of a partially dynamically reconfigurable FPGA architecture should be explored and methods to mitigate these should be devised.

Acknowledgements

First to my family for their love and support throughout my time in education and without whom I would not be here today.

To Lizzie for all her help in proof reading this Thesis.

To my supervisors Dr. Chris Clarke, Dr. Rob Watson and Dr. Adrian Evans for their support and supervision throughout this Ph.D. Particular thanks are given to Dr. Rob Watson for stepping in at the last minute.

To Intel FPGA and NVIDIA who kindly donated hardware for this research.

To the Engineering and Physical Science Research Council (EPSRC) for funding this research under grant number EP/M50645X/1.

References

[1] R. Meghana, "An overview of the 6th generation intel core processor (code-named skylake)," 2018, [Online]. Available: https://software.intel.com/en-us/articles/an-overview-of-the-6th-generation-intel-core-processor-code-named-skylake. Accessed: 27/11/2018.

[2] E. K. amd H. Moreton, N. Stam, and B. Bell, "NVIDIA turing architecture in-depth," 2018, [Online]. Available: https://devblogs.nvidia.com/nvidia-turing-architecture-in-depth/. Accessed: 27/11/2018.

[3] Intel, "Cyclone V FPGAS features," 2018, [Online]. Available: https://www.intel.co.uk/content/www/uk/en/products/programmable/fpga/cyclone-v/features.html. Accessed: 27/11/2018.

[4] Intel, "Intel ark," 2017, [Online]. Available: http://ark.intel.com/. Accessed: 23/11/2017.

[5] Intel, "The story of the Intel 4004 Intel's first microprocessor," [Online]. Available: https://www.intel.com/content/www/us/en/history/museum-story-of-intel-4004.html. Accessed: 23/11/2017.

[6] Intel, "The evolution of a revolution," [Online]. Available: http://download.intel.com/pressroom/kits/IntelProcessorHistory.pdf. Accessed: 23/11/2017.

[7] H. Xuejue, L. Wen-Chin, K. Charles, D. Hisamoto, C. Leland, J. Kedzierski, E. Anderson, H. Takeuchi, C. Yang-Kyu, K. Asano, V. Subramanian, K. Tsu-Jae, J. Bokor, and H. Chenming, "Sub 50-nm FinFET: PMOS," in *International Electron Devices Meeting. Technical Digest*, 1999, pp. 67–70.

[8] C. Walker and T. Campbell, "Intel completes acquisition of Altera," Tech. Rep., 28/12/15 2015, [Online]. Available: http://newsroom.altera.com/press-releases/nr-intel-acquisition-altera.htm.

[9] AMD, "AMD Ryzen desktop processors," 2018, [Online]. Available: https://www.amd.com/en/ryzen. Accessed: 08/03/2018.

[10] Intel, "Intel unveils the 8th gen intel core processor family for desktop, featuring intels best gaming processor ever," 2017, [Online]. Available: https://newsroom.intel.com/news-releases/intel-unveils-8th-gen-intel-core-processor-family-desktop/. Accessed: 26/01/2017.

[11] Intel, "8th gen intel core," 2017, [Online]. Available: https://newsroom.intel.com/press-kits/8th-gen-intel-core/. Accessed: 26/01/2017.

[12] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008, [Online]. Available: https://bitcoin.org/bitcoin.pdf. Accessed: 23/11/2017.

[13] Ethereum project, "A next-generation smart contract and decentralized application platform," 2017, [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper. Accessed: 23/11/2017.

[14] NVIDIA, "What is GPU computing?" 2017, [Online]. Available: http://www.nvidia.co.uk/object/cuda-parallel-computing-uk.html. Accessed: 23/11/2017.

[15] Khronos, "OpenCL overview," 2017, [Online]. Available: https://www.khronos.org/opencl/. Accessed: 23/11/2017.

[16] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, 1972.

[17] Intel, "Hardened floating-point processing in Arria 10 FPGAs and SoCs," [Online]. Available: https://www.altera.com/products/fpga/features/dsp/arria10-dsp-block.html. Accessed: 28/02/2018.

[18] G. E. Moore, "Cramming more components onto integrated circuits," *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 5, pp. 33–35, 2006, reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.

[19] G. Moore, "Are we really ready for VLSI^2?" in *IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, vol. XXII, 1979, pp. 54–55.

[20] G. E. Moore, "Lithography and the future of Moore's law," *IEEE Solid-State Circuits Society Newsletter*, vol. 20, no. 3, pp. 37–42, 2006.

[21] G. E. Moore, "No exponential is forever: but "forever" can be delayed! [semiconductor industry]," in *IEEE International Solid-State Circuits Conference. Digest of Technical Papers. ISSCC.*, 2003, pp. 20–23 vol.1.

[22] P. Matherat, D. Bouteaud, N. Forget, J. Lebrun, and J. P. Moreau, "A high-performance integrated true graphic processor," in *ESSCIRC: 6th European Solid State Circuits Conference*, 1980, pp. 271–273.

[23] C. K. Liu and C. M. Eastman, "Design of a graphic processor for computer-aided drafting," in *19th Design Automation Conference*, 1982, pp. 514–520.

[24] I. Nishimura, A. Shonaka, M. Morimoto, M. Asao, H. Mizukami, Y. Ohmari, O. Suzuki, and S. Yanase, "A color graphic processor for television broadcasting," *IEEE Transactions on Broadcasting*, vol. BC-29, no. 4, pp. 127–134, 1983.

[25] P. Geneste and D. Auger, "Real time graphics processor," in *[Proceedings] EURO ASIC '90*, 1990, pp. 314–316.

[26] Y. Suzuki and L. E. Atlas, "A comparison of processor topologies for a fast trainable neural network for speech recognition," in *International Conference on Acoustics, Speech, and Signal Processing*, 1989, pp. 2509–2512 vol.4.

[27] R. Ahrons, "Industrial research in microcircuitry at RCA: The early years, 1953-1963," *IEEE Annals of the History of Computing*, vol. 34, no. 1, pp. 60–73, 2012.

[28] T. S. Engine, "1963: Complementary MOS circuit configuration is invented," [Online]. Available: http://www.computerhistory.org/siliconengine/complementary-mos-circuit-configuration-is-invented/. Accessed: 30/11/2017.

[29] M. Lapedus, "What's after CMOS?" 20th Oct 2017 2014.

[30] L. Guo, L. Ye, C. Chen, Q. Huang, L. Yang, Z. Lv, X. An, and R. Huang, "Benchmarking TFET from a circuit level perspective: Applications and guideline," in *Circuits and Systems (ISCAS), IEEE International Symposium on.* IEEE, 2017, pp. 1–4.

[31] E. R. Hsieh, Y. C. Fan, K. Y. Chang, C. H. Chien, and S. S. Chung, "A novel design of P-N staggered face-tunneling TFET targeting for low power and appropriate performance applications," pp. 1–2, 24-27 April 2017 2017.

[32] R. C. Johnson, "FeFET to extend Moore's law," 20th Oct. 2017 2015.

[33] Y. Qin, Y. Xiong, K. Li, and M. Tang, "Simulation of FeFET-based basic logic circuits and current sense amplifier," in *Solid-State and Integrated Circuit Technology (ICSICT), 12th IEEE International Conference on.* IEEE, 2014, pp. 1–3.

[34] X. Mou, L. F. Register, and S. K. Banerjee, "Interplay among bilayer pseudospin field-effect transistor (BiSFET) performance, BiSFET scaling and condensate strength," in *Simulation of Semiconductor Processes and Devices (SISPAD), International Conference on.* IEEE, 2014, pp. 309–312.

[35] X. Mou, L. F. Register, A. H. MacDonald, and S. K. Banerjee, "Bilayer pseudospin junction transistor (BiSJT) for "beyond-CMOS" logic," *IEEE Transactions on Electron Devices*, vol. PP, no. 99, pp. 1–4, 2017.

[36] H. Xuejue, L. Wen-Chin, C. Kuo, D. Hisamoto, C. Leland, J. Kedzierski, E. Anderson, H. Takeuchi, C. Yang-Kyu, K. Asano, V. Subramanian, K. Tsu-Jae, J. Bokor, and H. Chenming, "Sub-50 nm p-channel FinFET," *IEEE Transactions on Electron Devices*, vol. 48, no. 5, pp. 880–886, 2001.

[37] V. Stojanovi, A. Joshi, C. Batten, Y. J. Kwon, S. Beamer, S. Chen, and K. Asanovi, "Design-space exploration for CMOS photonic processor networks," in *Conference on Optical Fiber Communication (OFC/NFOEC), collocated National Fiber Optic Engineers Conference*, 2010, pp. 1–3.

[38] V. Stojanovi, A. Joshi, C. Batten, Y. J. Kwon, S. Beamer, S. Chen, and K. Asanovi, "CMOS photonic processor-memory networks," in *IEEE Photonics Society Winter Topicals Meeting Series (WTM)*, 2010, pp. 118–119.

[39] P. K. Shen, C. T. Chen, C. H. Chang, C. Y. Chiu, S. L. Li, C. C. Chang, and M. L. Wu, "Implementation of chip-level optical interconnect with laser and photodetector using SOI-based 3-D guided-wave path," *IEEE Photonics Journal*, vol. 6, no. 6, pp. 1–10, 2014.

[40] Y. Lin, J. Hao, D. Jianfeng, and Z. Lei, "Silicon optical matrix processor for parallel computing," in *Progress in Electromagnetic Research Symposium (PIERS)*, 2016, pp. 791–791.

[41] Q. Vinckier, A. Bouwens, M. Haelterman, and S. Massar, "Autonomous all-photonic processor based on reservoir computing paradigm," in *Conference on Lasers and Electro-Optics (CLEO)*, 2016, pp. 1–2.

[42] Z. Xiao and B. M. Baas, "Processor tile shapes and interconnect topologies for dense on-chip networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 6, pp. 1377–1390, 2014.

[43] O. F. Yousif, M. H. Salih, R. B. Ahmed, L. A. K. Hasnawi, and R. K. Al-Janabi, "FPGA based embedded homogenous and hetrogenous multi-processor SoC design: A review," in *IEEE Conference on Open Systems (ICOS)*, 2014, pp. 99–104.

[44] R. Soleymanpour, S. Mohammadi, and H. Rajabi, "A synthesis algorithm for customized heterogeneous multi-processors," in *International SoC Design Conference (ISOCC)*, 2012, pp. 151–154.

[45] S. Sarma and N. Dutt, "Cross-layer exploration of heterogeneous multicore processor configurations," in *28th International Conference on VLSI Design*, 2015, pp. 147–152.

[46] Z. Xiao and B. Baas, "A hexagonal shaped processor and interconnect topology for tightly-tiled many-core architecture," in *IEEE/IFIP 20th International Conference on VLSI and System-on-Chip (VLSI-SoC)*, 2012, pp. 153–158.

[47] M. Nakajima, T. Yamamoto, M. Yamasaki, K. Kaneko, and T. Hosoki, "Homogenous dual-processor core with shared L1 cache for mobile multimedia SoC," in *IEEE Symposium on VLSI Circuits*, 2007, pp. 216–217.

[48] S. Sarma and N. Dutt, "Cross-layer exploration of heterogeneous multicore processor configurations,," in *28th International Conference on VLSI Design*, 2015, pp. 147–152.

[49] J. D. Souza, L. Carro, M. B. Rutzig, and A. C. S. Beck, "A reconfigurable heterogeneous multicore with a homogeneous ISA," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016, pp. 1598–1603.

[50] Y. Ge and Q. Qiu, "Task allocation for minimum system power in a homogenous multicore processor," in *International Conference on Green Computing*, 2010, pp. 299–306.

[51] P. Xiang, Y. Yang, M. Mantor, N. Rubin, and H. Zhou, "Revisiting ILP designs for throughput-oriented GPGPU architecture," in *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2015, pp. 121–130.

[52] M. P. Wachowiak, M. C. Timson, and D. J. DuVal, "Adaptive particle swarm optimization with heterogeneous multicore parallelism and GPU acceleration," *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, pp. 1–1, 2017.

[53] K. Taek-Jun, J. Sondeen, and J. Draper, "Floating-point division and square root using a taylor-series expansion algorithm," in *50th Midwest Symposium on Circuits and Systems*, 2007, pp. 305–308.

[54] W. Liang-Kai and M. J. Schulte, "Decimal floating-point square root using Newton-Raphson iteration," in *IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP)*, 2005, pp. 309–315.

[55] M. A. Cornea-Hasegan, R. A. Golliver, and P. Markstein, "Correctness proofs outline for Newton-Raphson based floating-point divide and square root algorithms," in *Proceedings 14th IEEE Symposium on Computer Arithmetic*, 1999, pp. 96–105.

[56] P. Kachhwal and B. C. Rout, "Novel square root algorithm and its FPGA implementation," in *International Conference on Signal Propagation and Computer Technology (ICSPCT)*, 2014, pp. 158–162.

[57] A. Amaricai and O. Boncalo, "FPGA implementation of very high radix square root with prescaling," in *19th IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, 2012, pp. 221–224.

[58] A. Hasnat, T. Bhattacharyya, A. Dey, S. Halder, and D. Bhattacharjee, "A fast FPGA based architecture for computation of square root and inverse square root," in *Devices for Integrated Circuit (DevIC)*, 2017, pp. 383–387.

[59] L. Yamin and C. Wanming, "A new non-restoring square root algorithm and its VLSI implementations," in *Proceedings International Conference on Computer Design. VLSI in Computers and Processors*, 1996, pp. 538–544.

[60] R. V. W. Putra and T. Adiono, "A register-free and homogenous architecture for square root algorithm," in *Computer, Control, Informatics and Its Applications (IC3INA), International Conference on.* IEEE, 2014, pp. 64–68.

[61] R. V. W. Putra, "A novel fixed-point square root algorithm and its digital hardware design," in *International Conference on ICT for Smart Society*, 2013, pp. 1–4.

[62] K. N. Vijeyakumar, V. Sumathy, P. Vasakipriya, and A. D. Babu, "FPGA implementation of low power high speed square root circuits," in *IEEE International Conference on Computational Intelligence and Computing Research*, 2012, pp. 1–5.

[63] J. E. Volder, "The CORDIC trigonometric computing technique," *IRE Transactions on Electronic Computers*, vol. EC-8, no. 3, pp. 330–334, Sept 1959.

[64] J. E. Volder, "The birth of CORDIC," *The Journal of VLSI Signal Processing-Systems for Signal, Image, and Video Technology*, vol. 25, pp. 101–105, June 2000.

[65] R. V. W. Putra and T. Adiono, "Optimized hardware algorithm for integer cube root calculation and its efficient architecture," in *International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS)*, 2015.

[66] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," *Micro, IEEE*, vol. 35, no. 3, pp. 10–22, 2015.

[67] T. Liu, Q. Wang, X. Wang, and F. Gao, "Pipeline-based parallel framework for mass file processing," in *International Conference on Cloud and Service Computing*, 2013, pp. 42–48.

[68] B. Shinde and S. T. Singh, "Data parallelism for distributed streaming applications," in *International Conference on Computing Communication Control and automation (ICCUBEA)*, 2016, pp. 1–4.

[69] E. Sitaridi, "Hardware acceleration of database analytics," in *IEEE 33rd International Conference on Data Engineering (ICDE)*, 2017, pp. 1616–1616.

[70] S. Suresh, S. F. Beldianu, and S. G. Ziavras, "FPGA and ASIC square root designs for high performance and power efficiency," in *IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, 2013, pp. 269–272.

[71] D. G. Perera and L. Kin Fun, "FPGA-based reconfigurable hardware for compute intensive data mining applications," in *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), International Conference on*, 2011, pp. 100–108.

[72] G. Hegde, Siddhartha, N. Ramasamy, V. Buddha, and N. Kapre, "Evaluating embedded FPGA accelerators for deep learning applications," in *IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016, pp. 25–25.

[73] A. M. Lalge, A. Shrivastav, and S. U. Bhandari, "Implementing PSK modems on FPGA using partial reconfiguration," in *Computing Communication Control and Automation (ICCUBEA), International Conference on*, 2015, pp. 917–921.

[74] S. Mosharafa, G. Ebrahim, and A. Zekry, "A novel algorithm for synchronizing audio and video streams in MPEG-2 system layer," pp. 142–147, 2014.

[75] A. C. Bovik, T. S. Huang, and J. Munson, D. C., "A generalization of median filtering using linear combinations of order statistics," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 31, no. 6, pp. 1342–1350, 1983.

[76] M. Jahiruzzaman, S. Saha, and M. A. K. Hawlader, "Dynamically reconfigurable parallel architecture implementation of 2D convolution for image processing over FPGA," in *Electrical Engineering and Information Communication Technology (ICEEICT), International Conference on*, 2015, pp. 1–6.

[77] L. Ye, Y. Qingming, T. Bin, and X. Wencong, "Fast double-parallel image processing based on FPGA," in *Vehicular Electronics and Safety (ICVES), IEEE International Conference on*, 2011, pp. 97–102.

[78] J. Duan, Y. Deng, and K. Liang, "Development of image processing system based on DSP and FPGA," in *Electronic Measurement and Instruments, ICEMI. 8th International Conference on*, 2007, pp. 2–791–2–794.

[79] T. Q. Pham and L. J. van Vliet, "Separable bilateral filtering for fast video preprocessing," in *Multimedia and Expo, ICME IEEE International Conference on*, 2005, p. 4.

[80] H. S. Neoh and A. Hazanchuk, "Adaptive edge detection for real-time video processing using FPGAs," *Global Signal Processing*, vol. 7, no. 3, pp. 2–3, 2004.

[81] L. Goddard and I. Stephenson, "Hardware accelerated shaders using FPGAs," in *TPCG*.

[82] L. Middendorf, F. Mühlbauer, G. Umlauf, and C. Bobda, "Embedded vertex shader in fpga," in *Embedded System Design: Topics, Techniques and Trends*, A. Rettberg, M. C. Zanella, R. Dömer, A. Gerstlauer, and F. J. Rammig, Eds. Boston, MA: Springer US, 2007, pp. 155–164.

[83] K. Kyungsu, L. Hoosung, C. Seonghyun, and P. Seongmo, "Implementation of 3D graphics accelerator using full pipeline scheme on FPGA," in *International SoC Design Conference*, vol. 02, 2008, pp. II–97–II–100.

[84] Y. Liu, "A novel mesa-based OpenGL implementation on an FPGA-based embedded system," in *International Conference on Audio, Language and Image Processing*, 2014, pp. 78–83.

[85] R. T. Gu, T. C. Yeh, W. S. Hunag, T. Y. Huang, C. H. Tsai, C. N. Lee, M. C. Chiang, S. F. Hsiao, Y. N. Chang, and I. J. Huang, "A low cost tile-based 3D graphics full pipeline with real-time performance monitoring support for OpenGL ES in consumer electronics," in *IEEE International Symposium on Consumer Electronics*, 2007, pp. 1–6.

[86] L. Middendorf and C. Haubelt, "A novel graphics processor architecture based on partial stream rewriting," in *Design and Architectures for Signal and Image Processing (DASIP), Conference on*, 2013, pp. 38–45.

[87] TES Electronic Solutions, "D/AVE NX - preliminary product brief," Tech. Rep., August 9th 2016.

[88] Y. Rui, S. Yanmei, H. Kun, and Y. Yang, "Online evolution of image filters based on dynamic partial reconfiguration of FPGA," in *Natural Computation (ICNC), 11th International Conference on*, 2015, pp. 999–1005.

[89] G. Li, D. Chen, D. Wang, and D. Zhang, "Task clustering and scheduling to multiprocessors with duplication," in *Proceedings International Parallel and Distributed Processing Symposium*, 2003, p. 8 pp.

[90] R. Ferreira, W. Denver, M. Pereira, S. Wong, C. A. Lisboa, and L. Carro, "A dynamic modulo scheduling with binary translation: Loop optimization with software compatibility," *Journal of Signal Processing Systems for Signal Image and Video Technology*, vol. 85, no. 1, pp. 45–66, 2016.

[91] A. Prakash, S. K. Lam, C. T. Clarke, and T. Srikanthan, "Instruction set customization for area-constrained FPGA designs," in *IEEE International SOC Conference*, 2011, pp. 329–334.

[92] A. Prakash, C. T. Clarke, and T. Srikanthan, "Custom instructions with local memory elements without expensive DMA transfers," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, 2012, pp. 647–650.

[93] A. Prakash, S. K. Lam, T. Srikanthan, and C. T. Clarke, "Modelling communication overhead for accessing local memories in hardware accelerators," in *IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, 2013, pp. 31–34.

[94] X. Di, S. Fazhuang, D. Zhantao, and H. Wei, "A design flow for FPGA partial dynamic reconfiguration," in *Instrumentation, Measurement, Computer, Communication and Control (IMCCC), Second International Conference on*, 2012, Conference Proceedings, pp. 119–123.

[95] O. Diessel, "Opportunities and challenges for dynamic FPGA reconfiguration in electronic measurement and instrumentation," in *IEEE 11th International Conference on Electronic Measurement & Instruments*, vol. 1, 2013, pp. 258–263.

[96] D. Pagano, M. Vuka, M. Rabozzi, R. Cattaneo, D. Sciuto, and M. D. Santambrogio, "Thermal-aware floorplanning for partially-reconfigurable FPGA-based systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2015, pp. 920–923.

[97] T. F. Oliver and D. L. Maskell, "Execution objects for dynamically reconfigurable FPGA systems," in *Field Programmable Logic and Applications, FPL. International Conference on*, 2006, pp. 1–4.

[98] N. I. Rafla and D. Gauba, "Hardware implementation of context switching for hard real-time operating systems," in *IEEE 54th International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2011, pp. 1–4.

[99] L. Sawalha, M. P. Tull, and R. D. Barnes, "Hardware thread-context switching," *Electronics Letters*, vol. 49, no. 6, pp. 389–391, 2013.

[100] F. Liu, F. Guo, Y. Solihin, S. Kim, and A. Eker, "Characterizing and modeling the behavior of context switch misses!" in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008, pp. 91–101.

[101] Z. Lin, L. Nyland, and H. Zhou, "Enabling efficient preemption for SIMT architectures with lightweight context switching," in *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 898–908.

[102] D. G. Perera and L. Kin Fun, "Similarity computation using reconfigurable embedded hardware," in *Dependable, Autonomic and Secure Computing, DASC. Eighth IEEE International Conference on*, 2009, pp. 323–329.

[103] U. Langenbach, S. Wiehler, and E. Schubert, "Evaluation of a declarative linux kernel FPGA manager for dynamic partial reconfiguration," in *International Conference on FPGA Reconfiguration for General-Purpose Computing (FPGA4GPC)*, 2017, pp. 13–18.

[104] E. Cetin, O. Diessel, G. Lingkan, and V. Lai, "Towards bounded error recovery time in FPGA-based TMR circuits using dynamic partial reconfiguration," in *Field Programmable Logic and Applications (FPL), 23rd International Conference on*, 2013, pp. 1–4.

[105] A. L. Silva, M. C. Lorencena, R. Ribeiro, M. A. C. Barbosa, and M. Teixeira, "Supervisory control of multiple robots subject to context switching," in *12th IEEE International Conference on Industry Applications (INDUSCON)*, 2016, pp. 1–7.

[106] Y. Dong, J. Mao, H. Guan, J. Li, and Y. Chen, "A virtualization solution for BYOD with dynamic platform context switching," *IEEE Micro*, vol. 35, no. 1, pp. 34–43, 2015.

[107] K. T. Durkee, C. Shabarekh, C. Jackson, and G. Ganberg, "Flexible autonomous support to aid context and task switching," in *IEEE International Multi-Disciplinary Conference on Cognitive Methods in Situation Awareness and Decision Support (CogSIMA)*, 2011, pp. 204–207.

[108] E. Chalmers, E. B. Contreras, B. Robertson, A. Luczak, and A. Gruber, "Context-switching and adaptation: Brain-inspired mechanisms for handling environmental changes," in *2016 International Joint Conference on Neural Networks (IJCNN)*, 2016, pp. 3522–3529.

[109] M. Feilen, A. Iliopoulos, M. Ihmig, and W. Stechele, "Partitioning and context switching for a reconfigurable FPGA-based DAB receiver," in *Proceedings of the Conference on Design and Architectures for Signal and Image Processing*, Conference Proceedings, pp. 1–8.

[110] L. Santiago, L. A. J. Marzulo, A. C. Sena, T. A. O. Alves, and F. M. G. Frana, "Optimising loops in dynamic dataflow," *IET Circuits, Devices & Systems*, vol. 11, no. 2, pp. 113–122, 2017.

[111] T. Loke and J. B. Wang, "OptQC v1.3: An (updated) optimized parallel quantum compiler," *Computer Physics Communications*, vol. 207, pp. 531–532, 2016.

[112] S. Im and D. Shin, "OpenGL ESSL optimizing compiler for embedded 3D graphic processor," in *The 1st IEEE Global Conference on Consumer Electronics*, 2012, pp. 724–725.

[113] M. Mukherjee, A. Fell, and A. Guha, "DFGenTool: A dataflow graph generation tool for coarse grain reconfigurable architectures," in *30th International Conference on VLSI Design and 16th International Conference on Embedded Systems (VLSID)*, 2017, pp. 67–72.

[114] M. Ping, Z. Zhongyuan, S. Weiguang, and H. Weifeng, "An automatic parallelizer for coarse-grained reconfigurable processor," in *13th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, 2016, pp. 215–217.

[115] Y. Yankova, K. Bertels, S. Vassiliadis, R. Meeuws, and A. Virginia, "Automated HDL generation: Comparative evaluation," in *IEEE International Symposium on Circuits and Systems*, 2007, pp. 2750–2753.

[116] K. Bertels, G. Kuzmanov, E. M. Panainte, G. Gaydadjiev, Y. Yankova, V. M. Sima, K. Sigdel, R. Meeuws, and S. Vassiliadis, "Hartes toolchain early evaluation: Profiling, compilation and HDL generation," in *2007 International Conference on Field Programmable Logic and Applications*, 2007, pp. 402–408.

[117] L. Middendorf, C. Bobda, and C. Haubelt, "Hardware synthesis of recursive functions through partial stream rewriting," in *Design Automation Conference (DAC), 49th ACM/EDAC/IEEE*, 2012, pp. 1203–1211.

[118] Y. Kobayashi, S. Kobayashi, K. Okuda, K. Sakanushi, Y. Takeuchi, and M. Imai, "Synthesizable HDL generation method for configurable VLIW processors," in *ASP-DAC: Asia and South Pacific Design Automation Conference*, 2004, pp. 843–846.

[119] M. Lattuada, F. Ferrandi, and M. Perrotin, "Computer assisted design and integration of FPGA accelerators in aerospace systems," in *IEEE Aerospace Conference*, 2016, pp. 1–11.

[120] "taste," [Online]. Available: http://taste.tuxfamily.org/. Accessed: 06/03/2018.

[121] P. Team, "Bambu: A free framework for the high-level synthesis of complex applications," 2017, [Online]. Available: https://panda.dei.polimi.it/?page_id=31. Accessed: 31/10/2017.

[122] A. Romanov, M. Romanov, and A. Kharchenko, "FPGA-based control system reconfiguration using open source software," in *IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, 2017, pp. 976–981.

[123] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 3, pp. 19–20, 2007, reprinted from the AFIPS Conference Proceedings, Vol. 30, 1967.

[124] G. A. Constantinides, "Massively parallel numerical computation on FPGAs," in *2007 Institution of Engineering and Technology FPGA Developers' Forum*, Oct 2007, pp. 1–14.

[125] Xilinx, "Floating-point operator v7.1," 2018, [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_1/pg060-floating-point.pdf. Accessed: 22/11/2018.

[126] Intel, "Floating-point IP cores user guide," 2018, [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_altfp_mfug.pdf. Accessed: 22/11/2018.

[127] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, Jul. 2011.

[128] X. Fang and M. Lesser, "Open-source variable-precision floating-point library for major commercial FPGAs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 9, no. 3, pp. 1–17, July 2016.

[129] M. B. Kraeling, "Fixed-point math in time-critical C applications," in *WESCON96*, 1996, pp. 587–593.

[130] F. A. Nothaft, L. Fernandez, S. Cefali, N. Shah, J. Rael, and L. Darnell, "Pragma-based floating-to-fixed point conversion for the emulation of analog behavioral models," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2014, pp. 633–640.

[131] L. Gerlach, G. Pay-Vay, and H. Blume, "Efficient emulation of floating-point arithmetic on fixed-point SIMD processors," in *IEEE International Workshop on Signal Processing Systems (SiPS)*. IEEE, 2016, pp. 254–259.

[132] E. Manikandan, K. A. Karthigeyan, and K. I. A. James, "Design of parallel vector/scalar floating point co-processor for reconfigurable architecture," in *Computing, Electronics and Electrical Technologies (ICCEET), International Conference on*, 2012, pp. 841–845.

[133] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.

[134] Altera, "FPGA architecture," Report, 2006, [Online]. Available: https://www.altera.com/en_US/pdfs/literature/wp/wp-01003.pdf. Accessed: 13/03/2018.

[135] A. Beaumont-Smith, N. Burgess, S. Lefrere, and C. C. Lim, "Reduced latency IEEE floating-point standard adder architectures," in *Proceedings 14th IEEE Symposium on Computer Arithmetic (Cat. No.99CB36336)*, April 1999, pp. 35–42.

[136] A. Malik and S. Ko, "A study on the floating-point adder in FPGAs," in *2006 Canadian Conference on Electrical and Computer Engineering*, May 2006, pp. 86–89.

[137] S. Veeramachaneni and M. B. Srinivas, "Floating point adder/subtractor units realization by efficient arithmetic circuits," in *2015 11th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME)*, June 2015, pp. 244–246.

[138] S. Palekar and N. Narkhede, "High speed and area efficient single precision floating point arithmetic unit," in *2016 IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT)*, May 2016, pp. 1950–1954.

[139] P. Chatarasi, J. Shirako, and V. Sarkar, "Polyhedral optimizations of explicitly parallel programs," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, Oct 2015, pp. 213–226.

[140] M. Lu, J.-L. Wang, J. Wen, and X.-W. Dong, "Implementation of hodgkin-huxley neuron model in FPGAs," in *2016 Asia-Pacific International Symposium on Electromagnetic Compatibility (APEMC)*, vol. 01, May 2016, pp. 1115–1117.

[141] A. S. Cassidy, P. Merolla, J. V. Arthur, S. K. Esser, B. Jackson, R. Alvarez-Icaza, P. Datta, J. Sawada, T. M. Wong, V. Feldman, A. Amir, D. B. D. Rubin, F. Akopyan, E. McQuinn, W. P. Risk, and D. S. Modha, "Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores," in *International Joint Conference on Neural Networks (IJCNN)*, 2013, pp. 1–10.

[142] A. Banerjee, P. T. Wolkotte, R. D. Mullins, S. W. Moore, and G. J. M. Smit, "An energy and performance exploration of network-on-chip architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 3, pp. 319–329, March 2009.

[143] B. Land, "Simplified floating point for DSP," 2018, [Online]. Available: people.ece.cornell.edu/land/courses/ece5760/FloatingPoint/index.html. Accessed: 27/11/2018.

[144] Intel, "Floating-point IP cores user guide," 2015, [Online]. Available: https://www.altera.com/en_US/pdfs/literature/ug/archives/ug-altfp-mfug-15.0.pdf. Accessed: 06/02/2018.

[145] Sun Microsystems, "e_exp.c," 2004, [Online]. Available: http://www.netlib.org/fdlibm/e_exp.c. Accessed: 30/11/2017.

[146] W. A. Catterall, I. M. Raman, H. P. C. Robinson, T. J. Sejnowski, and O. Paulsen, "The Hodgkin-Huxley heritage: from channels to circuits." *The Journal of neuroscience : the official journal of the Society for Neuroscience*, vol. 32, no. 41, pp. 14 064–73, 2012.

[147] E. M. Izhikevich, "Simple model of spiking neurons," *IEEE Transactions on Neural Networks*, vol. 14, no. 6, pp. 1569–1572, 2003.

[148] Y. LeCun, C. Cortes, and C. J. Burge, "The MNIST database of handwritten digits," [Online]. Available: http://yann.lecun.com/exdb/mnist/. Accessed: 23/11/2017.

[149] R. R. Osorio, "Pipelined FPGA implementation of numerical integration of the Hodgkin-Huxley model," in *IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2016, pp. 202–206.

[150] APT Advanced Processor Technologies Research Group, "SpiNNaker home page," 2017, [Online]. Available: http://apt.cs.manchester.ac.uk/projects/SpiNNaker/. Accessed: 23/11/2017.

[151] A. E. Hindborg, P. Schleuniger, N. B. Jensen, and S. Karlsson, "Hardware realization of an FPGA processor - operating system call offload and experiences," in *Proceedings of the 2014 Conference on Design and Architectures for Signal and Image Processing*, 2014, pp. 1–8.

[152] A. Jafari, M. Ghovanloo, and T. Mohsenin, "A real-time embedded FPGA processor for a stand-alone dual-mode assistive device," pp. 199–199, 2017.

[153] V. Brost, C. Meunier, D. Saptono, and F. Yang, "Flexible VLIW processor based on FPGA for real-time image processing," in *Proceedings of the Conference on Design & Architectures for Signal & Image Processing (DASIP)*, 2011, pp. 1–8.

[154] C. Zima-Zegreanu, "Crash course in HLSL," p. A brief overview of HLSL, [Online]. Available: http://www.catalinzima.com/xna/tutorials/crash-course-in-hlsl/. Accessed: 10/03/2016.

[155] Intel, "Cyclone V FPGAs & SoCs," 2016, [Online]. Available: https://www.altera.com/products/fpga/cyclone-series/cyclone-v/overview.html. Accessed: 19/01/2016.

[156] S. Craitoiu, J. Popa, R. Postolache, and M. Krieger, "in2gpu," 2018, [Online]. Available: http://in2gpu.com/. Accessed: 17/03/2018.

[157] W. Tao, C. Chang Wen, and W. Changhu, "Barycentric coordinates based soft assignment for object classification," in *IEEE International Conference on Multimedia & Expo Workshops (ICMEW)*, 2016, pp. 1–6.

[158] G. Sellers, R. S. Wright Jr, and N. Haemel, *OpenGL SuperBible*, 7th ed. Addison-Wesley, 2016.

[159] A. L. Petrescu, F. Moldoveanu, V. Asavei, and A. Moldoveanu, "Virtual deferred rendering," in *20th International Conference on Control Systems and Computer Science*, 2015, pp. 373–378.

[160] S. Schneegans, F. Lauer, A. C. Bernstein, A. Schollmeyer, and B. Froehlich, "Guacamole - an extensible scene graph and rendering framework based on deferred shad-

ing," in *IEEE 7th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, 2014, pp. 35–42.

[161] NVIDIA, "TEGRA," 2017, [Online]. Available: http://www.nvidia.com/object/tegra-k1-processor.html. Accessed: 26/01/2017.

[162] Intel, "Cyclone V SoCs: Lowest system cost and power," 2017, [Online]. Available: https://www.altera.com/products/soc/portfolio/cyclone-v-soc/overview.html. Accessed: 26/01/2017.

[163] ARM, "Mali-400 ultra low power GPU," 2017, [Online]. Available: https://developer.arm.com/products/graphics-and-multimedia/mali-gpus/mali-400-gpu. Accessed: 30/06/2017.

[164] K. C. Kwan, X. Xu, L. Wan, T. T. Wong, and W. M. Pang, "Packing vertex data into hardware-decompressible textures," *IEEE Transactions on Visualization and Computer Graphics*, vol. PP, no. 99, pp. 1–1, 2018.

[165] NVIDIA, "NVIDIA home," 2016, [Online]. Available: http://www.nvidia.co.uk/page/home.html.

[166] M. Butts, "Future directions of dynamically reprogrammable systems," in *Proceedings of the IEEE 1995 Custom Integrated Circuits Conference*, May 1995, pp. 487–494.

[167] N. McKay, T. Melham, and K. W. Susanto, "Dynamic specialisation of XC6200 FPGAs by partial evaluation," in *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No.98TB100251)*, April 1998, pp. 308–309.

[168] C.-F. Wu and C.-W. Wu, "Testing interconnects of dynamic reconfigurable FPGAs," in *Proceedings of the ASP-DAC '99 Asia and South Pacific Design Automation Conference 1999 (Cat. No.99EX198)*, Jan 1999, pp. 279–282 vol.1.

[169] H. Kwok-Hay So, "Dynamic reconfiguration of Xilinx FPGAs," 2006, [Online]. Available: https://www.xilinx.com/univ/FPL06_Invited_Presentation_PLysaght.pdf. Accessed: 30/11/2017.

[170] Altera, "Increasing design functionality with partial and dynamic reconfiguration in 28-nm FPGAs," 2010, [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01137-stxv-dynamic-partial-reconfig.pdf. Accessed: 30/11/2017.

[171] VLSI n EDA, "Synchronizers," 2018, [Online]. Available: http://vlsiuniverse.blogspot.co.uk/2013/09/synchronization-schemes.html. Accessed: 19/03/2018.

[172] A. Bourge, O. Muller, and F. Rousseau, "Automatic high-level hardware checkpoint selection for reconfigurable systems," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2015, pp. 155–158.

[173] A. Beasley, L. Walker, and C. Clarke, "Developing and implementing dynamic partial reconfiguration for pre-emptible context switching and continuous end-to-end dataflow applications," pp. 1–10, Nov. 2015.

[174] M. X. Yue, D. Koch, and G. G. F. Lemieux, "Rapid overlay builder for xilinx FPGAs," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2015, pp. 17–20.

[175] Y. Wu and J. McAllister, "Architectural synthesis of multi-SIMD dataflow accelerators for FPGA," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 1, pp. 43–55, 2018.

[176] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2016.

[177] A. Canis, "LegUp high-level synthesis," 2017, [Online]. Available: http://legup.eecg.utoronto.ca/. Accessed: 31/10/2017.

[178] Xilinx, "Vivado high-level synthesis," 2017, [Online]. Available: https://www..com/products/design-tools/vivado/integration/esl-design.html. Accessed: 31/10/2017.

[179] Intel, "What is OpenCL?" 2017, [Online]. Available: https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html. Accessed: 31/10/2017.

[180] Mentor, "Handel-C synthesis methodology," [Online]. Available: https://www.mentor.com/products/fpga/handel-c/. Accessed: 27/04/2017.

[181] Impulse, "Optimize your FPGA applications," 2017, [Online]. Available: http://www.impulsec.com/. Accessed: 27/04/2017.

[182] Mentor, "High-level synthesis and RTL low-power," [Online]. Available: https://www.mentor.com/hls-lp/catapult-high-level-synthesis/c-systemc-hls. Accessed: 27/04/2017.

[183] J. a. M. P. Cardoso, P. C. Diniz, and M. Weinhardt, "Compiling for reconfigurable computing: A survey," *ACM Comput. Surv.*, vol. 42, no. 4, pp. 13:1–13:65, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1749603.1749604

[184] Intel, "Spectra-Q engine," 2015, [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/backgrounder/spectra-q-engine-backgrounder.pdf. Accessed: 30/11/2017.

[185] A. D. Brown, D. J. D. Milton, A. J. Rushton, and P. R. Wilson, "Behavioural synthesis utilising recursive definitions," *IET Computers & Digital Techniques*, vol. 6, no. 6, pp. 362–369, 2012.

[186] D. J. D. Milton, A. D. Brown, M. Zwolinski, and P. R. Wilson, "Behavioural synthesis utilising dynamic memory construct," in *IEE Proceedings - Computers and Digital Techniques*, vol. 151. IET, 2004, pp. 252–264.

Appendices

Appendix A

Resource use and performance for single and half precision implementations of floating point maths in hardware

A.1 Fundemental operators

Table A.1: Resource requirements and timing analysis for simple maths functions that do not require other functions to implement. Implementations are using single-precision floating-point accuracy.

Module	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs Recoverable by Dense Packing	[C] Estimate of ALMs Unavailable	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
Add/Subtract	499.0 (499.0)	712.0 (712.0)	216.5 (216.5)	3.5 (3.5)	561 (561)	1138 (1138)	0	162.15	155.38
Multiply	115.0 (115.0)	149.0 (149.0)	34.0 (34.0)	0.0 (0.0)	177 (177)	265 (265)	1	196.43	189.57
Multiply with no DSP	318.0 (85.5)	356.0 (136.1)	61.5 (58.1)	23.5 (7.5)	586 (113)	265 (265)	0	108.57	106.33
Greater than	57.5 (57.5)	66.0 (66.0)	8.5 (8.5)	0.0 (0.0)	98 (98)	98 (98)	0	214.73	214.22
Less than	58.5 (58.5)	64.0 (64.0)	5.5 (5.5)	0.0 (0.0)	98 (98)	98 (98)	0	209.86	210.93
Float to integer	102.0 (102.0)	119.0 (119.0)	17.0 (17.0)	0.0 (0.0)	136 (136)	133 (133)	0	230.79	232.99
Integer to float	156.0 (156.0)	182.0 (182.0)	26.0 (26.0)	0.0 (0.0)	246 (246)	171 (171)	0	137.19	131.93

Table A.2: Resource requirements and timing analysis for simple maths functions that do not require other functions to implement. Implementations are using half-precision floating-point accuracy.

Module	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs Recoverable by Dense Packing	[C] Estimate of ALMs Unavailable	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
Add/Subtract	255.0 (255.0)	364.0 (364.0)	109.0 (109.0)	0.0 (0.0)	285 (285)	588 (588)	0	243.31	246.73
Multiply	62.0 (62.0)	84.0 (84.0)	22.0 (22.0)	0.0 (0.0)	97 (97)	137 (137)	1	212.95	205.89
Multiply with no DSP	111.5 (61.2)	140.0 (88.9)	29.0 (28.1)	0.5 (0.3)	215 (99)	137 (137)	0	164.72	158.81
Greater than	32.5 (32.5)	33.5 (33.5)	1.0 (1.0)	0.0 (0.0)	56 (56)	50 (50)	0	265.46	261.71
Less than	33.5 (33.5)	34.5 (34.5)	1.0 (1.0)	0.0 (0.0)	56 (56)	50 (50)	0	248.76	245.1
Float to integer	45.5 (45.5)	56.5 (56.5)	11.0 (11.0)	0.0 (0.0)	69 (69)	69 (69)	0	353.98	353.36
Integer to float	67.0 (67.0)	86.0 (86.0)	19.0 (19.0)	0.0 (0.0)	110 (110)	90 (90)	0	268.17	259.88

A.2 Iterative operations

Table A.3: Resource requirements and timing analysis for hardware friendly implemenatations of a floating point square root operation. Implementations are using single-precision floating-point accuracy.

Module	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs Recoverable by Dense Packing	[C] Estimate of ALMs Unavailable	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
Traditional without pipeline	162.0 (162.0)	191.0 (191.0)	29.0 (29.0)	0.0 (0.0)	237 (237)	309 (309)	0	202.27	207.47
Traditional with pipeline	1142.5 (1142.5)	1801.0 (1801.0)	716.0 (716.0)	57.5 (57.5)	566 (566)	3769 (3769)	0	259.74	260.28
Proposed new design without pipeline	238.5 (238.5)	293.5 (293.5)	55.5 (55.5)	0.5 (0.5)	336 (336)	452 (452)	0	174.16	178.25
Proposed new design with pipeline	3584.5 (3584.5)	5719.0 (5719.0)	2355.5 (2355.5)	221.0 (221.0)	1913 (1913)	11564 (11564)	0	189.68	194.1

Table A.4: Resource requirements and timing analysis for hardware friendly implemenatations of a floating-point square-root operation. Implementations are using half-precision floating-point accuracy.

Module	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs Recoverable by Dense Packing	[C] Estimate of ALMs Unavailable	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
Traditional without pipeline	79.5 (79.5)	92.5 (92.5)	13.0 (13.0)	0.0 (0.0)	123 (123)	153 (153)	0	288.02	290.36
Traditional with pipeline	250.5 (250.5)	413.5 (413.5)	164.5 (164.5)	1.5 (1.5)	167 (167)	836 (836)	0	410.34	407.5
Proposed new design without pipeline	127.0 (127.0)	169.0 (169.0)	42.0 (42.0)	0.0 (0.0)	182 (182)	232 (232)	0	211.6	214.0
Proposed new design with pipeline	727.5 (727.5)	1230.5 (1230.5)	504.0 (504.0)	1.0 (1.0)	477 (477)	2465 (2465)	0	235.79	236.52

Table A.5: Resource requirements and timing analysis for floating-point invert and division operations using recursive a Newton Raphson approach to different numbers of iterations. Implementations are using single-precision floating-point accuracy.

Module	Iterations	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs Recoverable by Dense Packing	[C] Estimate of ALMs Unavailable	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
Newton-Raphson inversion	1	1577.0 (146.2)	2358.5 (175.4)	783.5 (29.2)	2.0 (0.0)	1571 (180)	4123 (133)	2	156.81	152.84
Newton-Raphson inversion	2	2562.0 (192.3)	3806.5 (231.7)	1246.5 (39.4)	2.0 (0.0)	2528 (254)	6719 (133)	4	159.69	153.63
Newton-Raphson inversion	3	3518.5 (227.1)	5196.5 (279.1)	1685.5 (51.9)	7.5 (0.0)	3463 (329)	9315 (133)	6	161.66	155.01
Newton-Raphson inversion	4	4499.0 (285.5)	6642.5 (331.8)	2150.0 (46.3)	6.5 (0.0)	4398 (404)	11911 (133)	8	163.05	156.32
Newton-Raphson inversion	5	5463.0 (340.1)	8092.5 (388.1)	2637.0 (48.0)	7.5 (0.0)	5333 (479)	14507 (133)	10	159.21	153.14
Newton-Raphson inversion	6	6446.0 (396.7)	9564.0 (445.9)	3133.5 (49.2)	15.5 (0.0)	6268 (554)	17103 (133)	12	155.45	149.97
Newton-Raphson inversion	7	7391.0 (441.5)	10988.0 (484.6)	3606.0 (43.1)	9.0 (0.0)	7203 (629)	19699 (133)	14	157.85	151.22
Newton-Raphson inversion	8	8315.0 (492.5)	12448.5 (539.8)	4144.0 (47.3)	10.5 (0.0)	8138 (704)	22295 (133)	16	161.84	154.73
Newton-Raphson inversion	9	9319.5 (529.6)	13823.0 (597.1)	4519.0 (67.5)	15.5 (0.0)	9073 (779)	24891 (133)	18	157.78	152.07
Newton-Raphson inversion	10	10283.5 (553.2)	15176.0 (618.5)	4904.5 (65.3)	12.0 (0.0)	10008 (854)	27487 (133)	20	157.58	150.81
Division	1	1928.0 (100.3)	2883.5 (106.0)	961.5 (5.7)	6.0 (0.0)	1764 (152)	5303 (0)	3	156.25	152.79
Division	2	3052.5 (153.3)	4618.0 (155.0)	1572.0 (1.7)	6.5 (0.0)	2755 (226)	8427 (0)	5	158.45	158.98
Division	3	4165.5 (199.7)	6303.0 (213.3)	2149.0 (13.7)	11.5 (0.0)	3722 (301)	11551 (0)	7	162.65	155.26
Division	4	5263.0 (250.5)	7935.0 (264.5)	2678.0 (14.0)	6.0 (0.0)	4689 (376)	14675 (0)	9	159.95	152.98
Division	5	6382.5 (304.2)	9637.0 (315.3)	3265.5 (11.1)	11.0 (0.0)	5652 (451)	17799 (0)	11	158.28	153.87
Division	6	7490.0 (353.2)	11382.0 (369.7)	3912.5 (16.5)	20.5 (0.0)	6623 (526)	20923 (0)	13	159.82	153.68
Division	7	8603.5 (403.4)	13099.0 (424.0)	4514.5 (20.7)	19.0 (0.0)	7590 (601)	24047 (0)	15	161.37	154.63
Division	8	9693.0 (453.2)	14727.0 (473.3)	5048.0 (20.2)	14.0 (0.0)	8557 (676)	27171 (0)	17	155.47	150.76
Division	9	10795.0 (505.0)	16408.5 (517.7)	5642.5 (12.7)	29.0 (0.0)	9524 (751)	30295 (0)	19	163.11	156.79
Division	10	11925.5 (555.6)	18000.5 (571.7)	6098.5 (16.1)	23.5 (0.0)	10492 (826)	33419 (0)	21	158.18	151.68

Table A.6: Resource requirements and timing analysis for floating-point invert and division operations using recursive a Newton-Raphson approach to different numbers of iterations. Implementations are using half-precision floating-point accuracy.

Module	Iterations	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs Recoverable by Dense Packing	[C] Estimate of ALMs Unavailable	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
Newton-Raphson inversion	1	837.5 (57.2)	1200.5 (66.3)	364.5 (9.1)	1.5 (0.0)	862 (67)	2127 (69)	2	207.77	201.78
Newton-Raphson inversion	2	1356.5 (74.8)	1946.5 (86.7)	596.0 (11.8)	6.0 (0.0)	1388 (91)	3469 (69)	4	195.85	190.11
Newton-Raphson inversion	3	1868.5 (89.7)	2696.0 (100.3)	840.0 (10.7)	12.5 (0.0)	1915 (115)	4811 (69)	6	193.57	187.69
Newton-Raphson inversion	4	2364.0 (109.8)	3428.5 (134.0)	1076.0 (24.2)	11.5 (0.0)	2351 (146)	6153 (69)	8	193.42	196.0
Newton-Raphson inversion	5	2884.0 (130.7)	4139.0 (150.2)	1269.0 (19.5)	14.0 (0.0)	2858 (171)	7495 (69)	10	198.65	198.49
Newton-Raphson inversion	6	3398.0 (148.4)	4911.0 (172.4)	1528.5 (24.0)	15.5 (0.0)	3365 (196)	8837 (69)	12	203.96	197.55
Newton-Raphson inversion	7	3901.5 (161.0)	5601.5 (192.6)	1702.0 (31.6)	2.0 (0.0)	3872 (221)	10179 (69)	14	201.01	201.21
Newton-Raphson inversion	8	4416.0 (185.7)	6427.0 (202.9)	2019.0 (17.2)	8.0 (0.0)	4379 (246)	11521 (69)	16	203.0	196.7
Newton-Raphson inversion	9	4942.0 (204.2)	7147.0 (222.6)	2223.0 (18.3)	18.0 (0.0)	4886 (271)	12863 (69)	18	201.9	199.28
Newton-Raphson inversion	10	5433.5 (223.4)	7829.5 (242.4)	2415.5 (19.0)	19.5 (0.0)	5393 (296)	14205 (69)	20	196.7	191.46
Division	1	1020.5 (33.8)	1492.5 (35.3)	473.0 (1.5)	1.0 (0.0)	998 (48)	2731 (0)	3	193.91	194.33
Division	2	1624.0 (49.6)	2369.5 (52.8)	748.5 (3.2)	3.0 (0.0)	1555 (72)	4345 (0)	5	200.12	200.32
Division	3	2210.0 (74.6)	3211.0 (75.8)	1010.0 (1.3)	9.0 (0.0)	2041 (102)	5959 (0)	7	201.05	200.92
Division	4	2808.5 (91.3)	4108.0 (91.8)	1307.5 (0.4)	8.0 (0.0)	2580 (127)	7573 (0)	9	203.21	196.93
Division	5	3397.0 (107.5)	4966.5 (111.7)	1585.0 (4.2)	15.5 (0.0)	3119 (152)	9187 (0)	11	200.92	195.05
Division	6	3974.5 (125.3)	5844.0 (132.0)	1880.5 (6.7)	11.0 (0.0)	3658 (177)	10801 (0)	13	193.61	188.36
Division	7	4580.0 (143.2)	6702.5 (150.3)	2157.0 (7.2)	34.5 (0.0)	4197 (202)	12415 (0)	15	193.76	188.57
Division	8	5180.0 (161.3)	7608.5 (168.7)	2454.0 (7.3)	25.5 (0.0)	4736 (227)	14029 (0)	17	191.2	191.31
Division	9	5795.5 (182.8)	8468.5 (188.3)	2709.0 (5.5)	36.0 (0.0)	5275 (252)	15643 (0)	19	193.09	186.74
Division	10	6366.5 (199.3)	9353.0 (203.8)	3005.0 (4.5)	18.5 (0.0)	5814 (277)	17257 (0)	21	199.44	197.28

A.3 Vector and matrix operators

Table A.7: Resource requirements and timing analysis for floating-point matrix and vector operations commonly performed by a GPU. Implementations are using single-precision floating-point accuracy.

Module	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs Recoverable by Dense Packing	[C] Estimate of ALMs Unavailable	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
Normalise	6524.5 (265.3)	10130.5 (327.6)	3828.0 (68.6)	222.0 (6.3)	4976 (349)	19034 (292)	6	156.91	150.6
Dot product	913.5 (119.7)	1251.5 (143.0)	349.5 (29.7)	11.5 (6.3)	1121 (165)	2066 (133)	3	163.24	159.01
Vector Length	4417.0 (114.6)	6919.0 (126.7)	2673.0 (15.3)	171.0 (3.2)	3014 (161)	13473 (102)	3	156.64	158.53

Table A.8: Resource requirements and timing analysis for floating-point matrix and vector operations commonly performed by a GPU. Implementations are using half-precision floating-point accuracy.

Module	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs Recoverable by Dense Packing	[C] Estimate of ALMs Unavailable	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
Normalise	2353.0 (134.2)	3496.0 (168.5)	1163.5 (35.8)	20.5 (1.5)	2177 (184)	6481 (180)	6	178.95	177.62
Dot product	501.0 (75.5)	687.5 (78.3)	187.0 (2.8)	0.5 (0.0)	643 (95)	1084 (85)	3	190.88	186.05
Vector Length	1236.0 (70.1)	1897.5 (71.3)	669.5 (2.1)	8.0 (1.0)	1114 (92)	3511 (70)	3	200.4	199.44

Appendix B

Floating point adders and multipliers in single and half precision

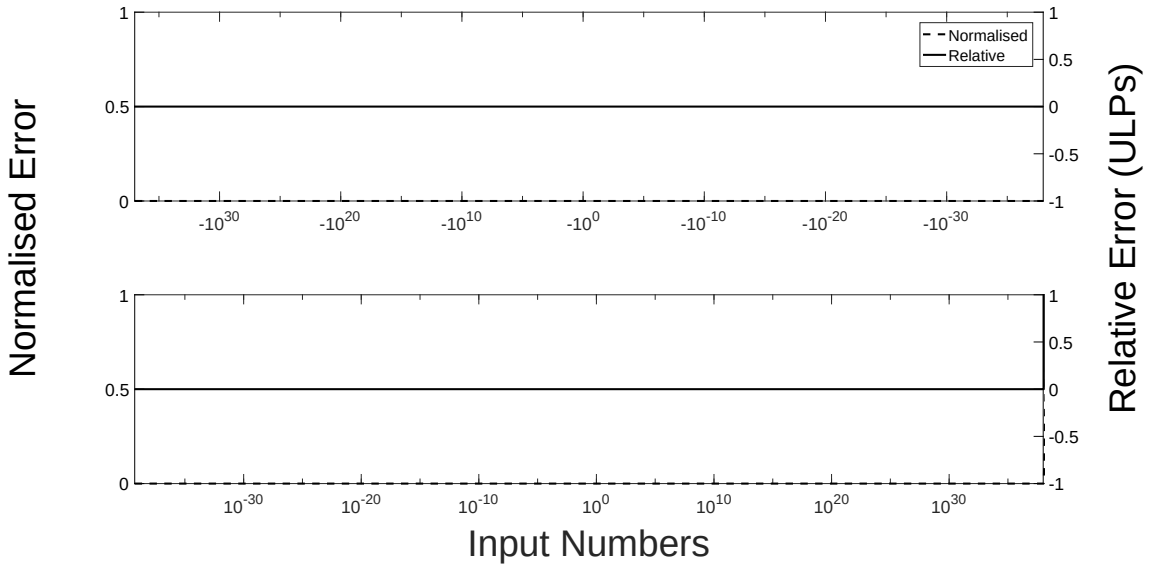


Figure B.1: Relative and absolute error for floating-point adder in single-precision

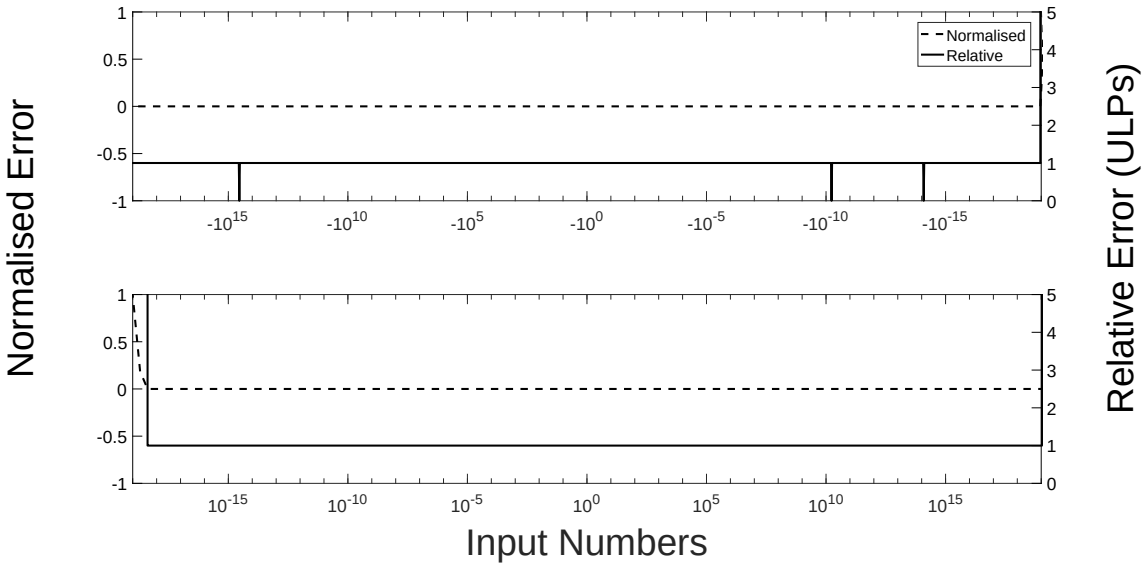


Figure B.2: Relative and absolute error for floating-point multipler in single-precision

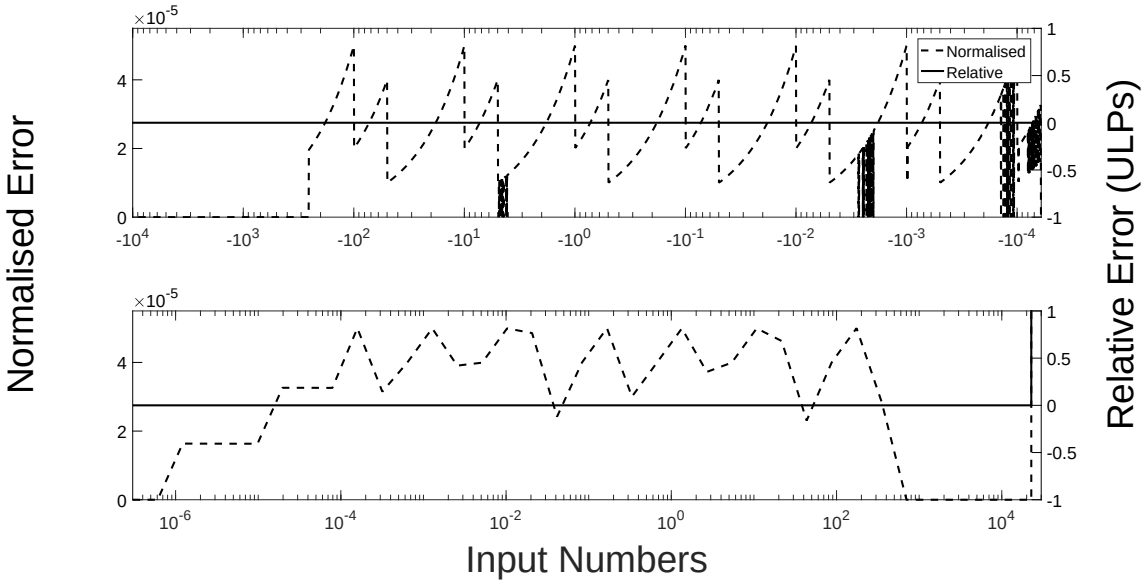


Figure B.3: Relative and absolute error for floating-point adder in half-precision

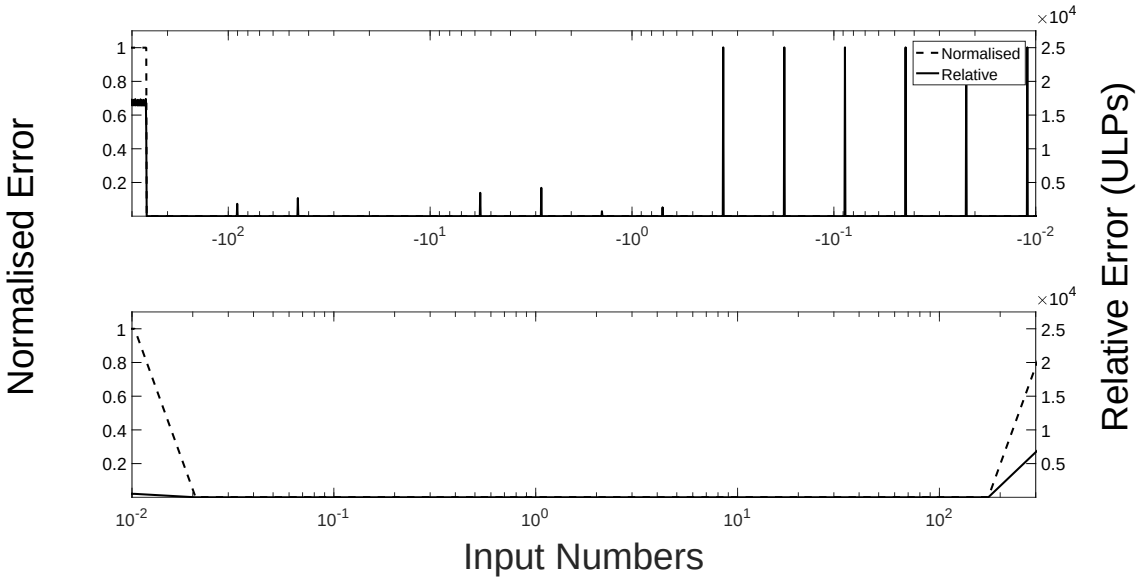


Figure B.4: Relative and absolute error for floating-point muliplier in half-precision

Appendix C

Accuracy of Newton-Raphson inversion algorithm implemented in hardware for IEEE-754R standard input formats

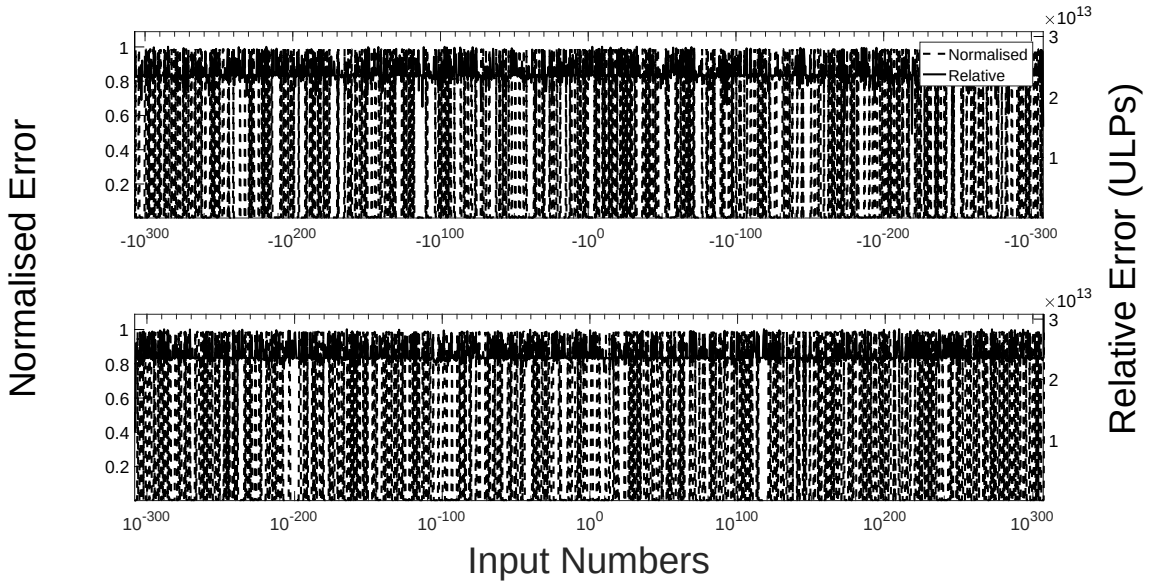


Figure C.1: A double-precision Newton-Raphson iteration-based inverter realised in hardware with only a single NR stage. Top half of the graph shows error for a negative input, bottom half shows error for a positive input.

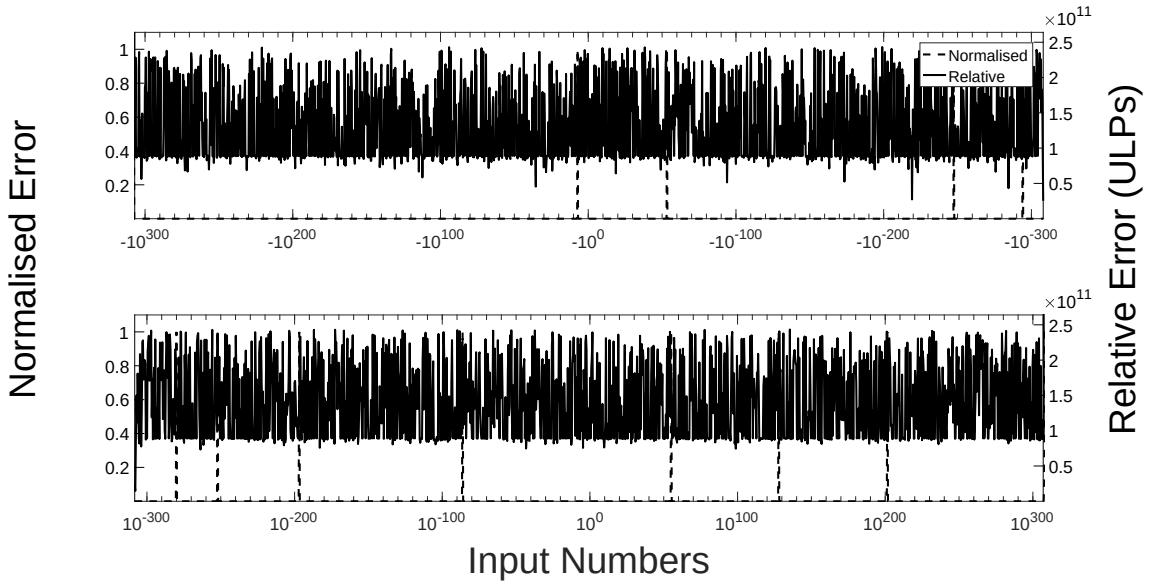


Figure C.2: A double-precision Newton-Raphson iteration-based inverter realised in hardware with only two NR stages. Top half of the graph shows error for a negative input, bottom half shows error for a positive input.

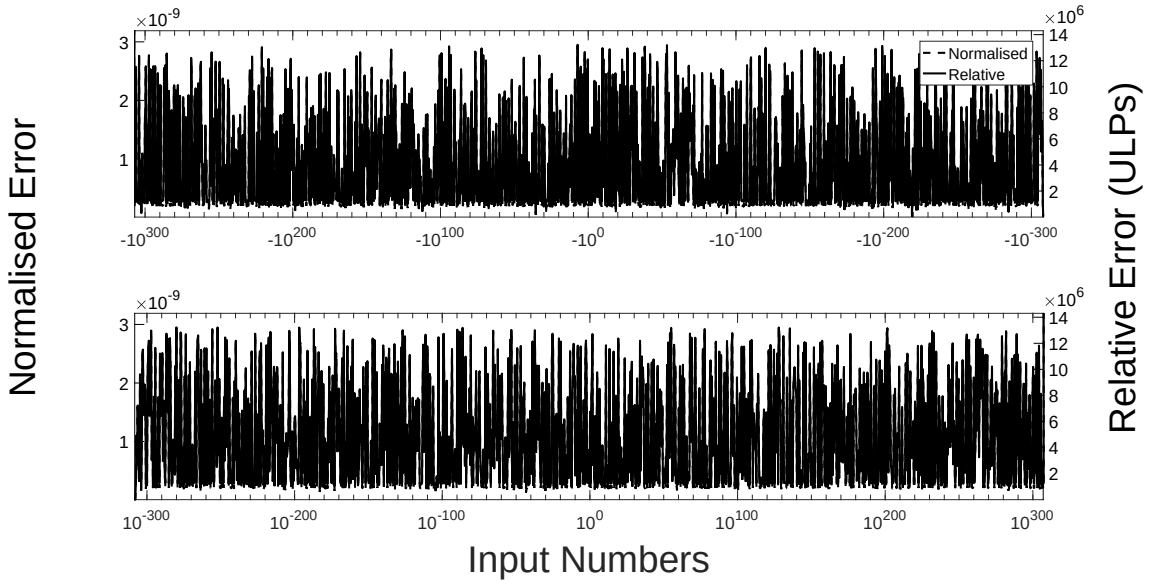


Figure C.3: A double-precision Newton-Raphson iteration-based inverter realised in hardware with only three NR stages. Top half of the graph shows error for a negative input, bottom half shows error for a positive input.

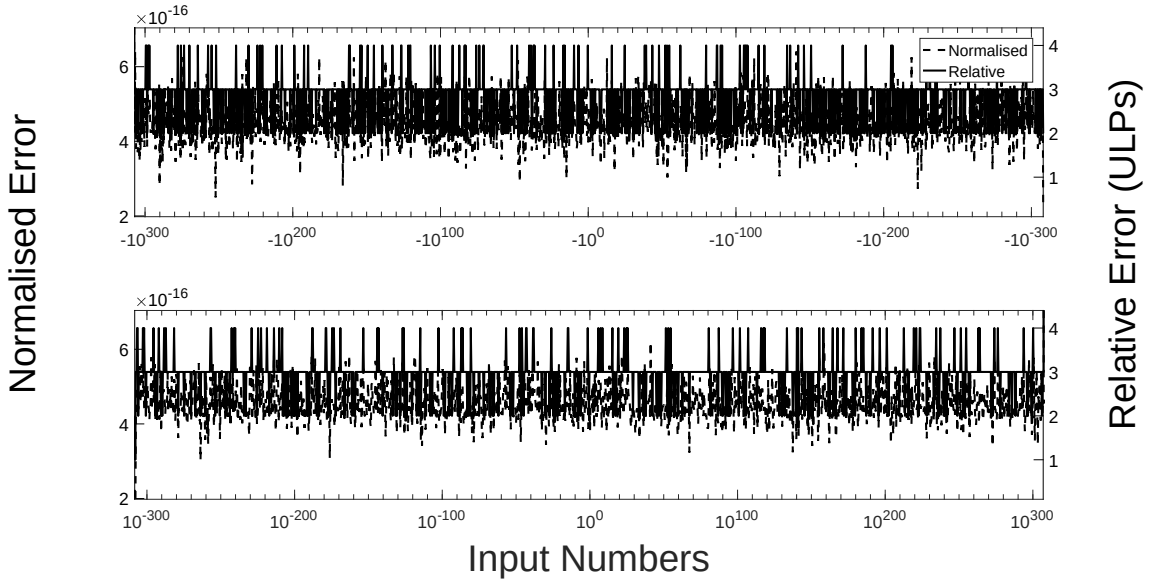


Figure C.4: A double-precision Newton-Raphson iteration-based inverter realised in hardware with only four NR stages. Top half of the graph shows error for a negative input, bottom half shows error for a positive input.

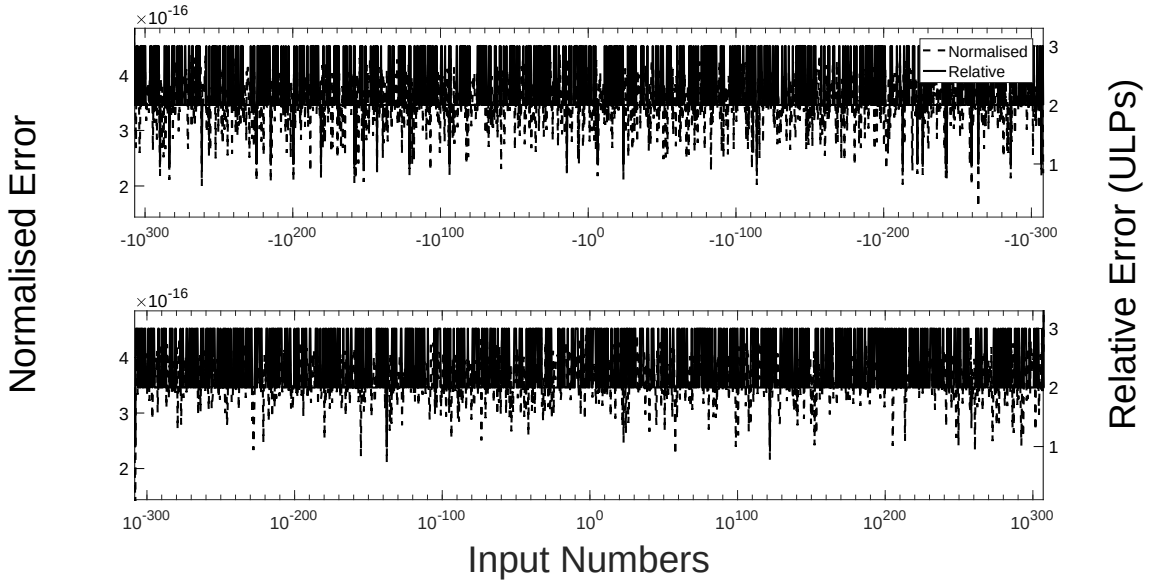


Figure C.5: A double-precision Newton-Raphson iteration-based inverter realised in hardware with only five NR stages. Top half of the graph shows error for a negative input, bottom half shows error for a positive input.

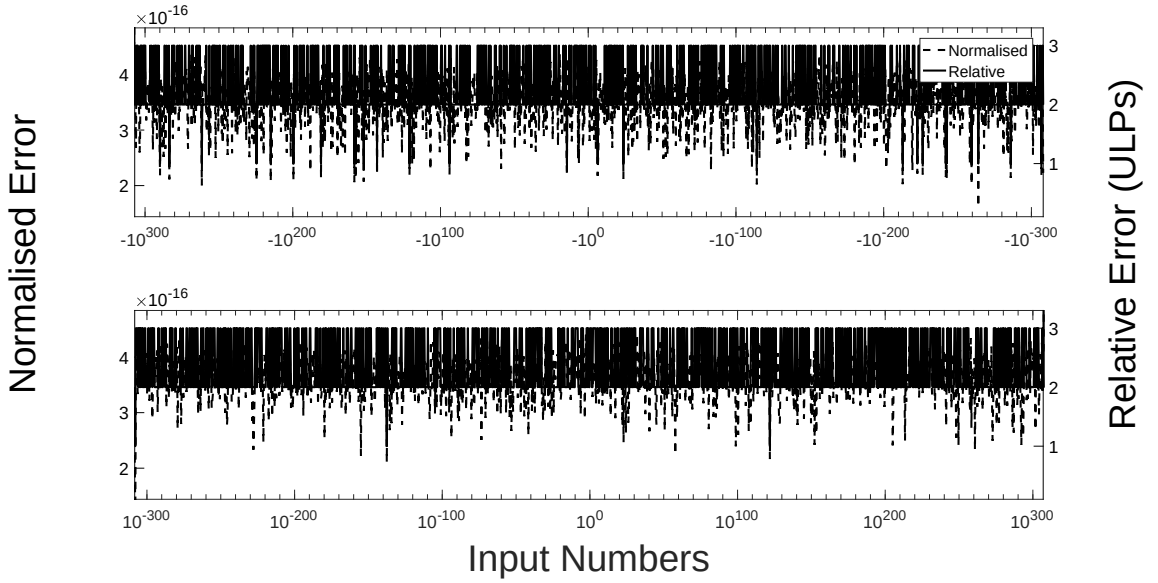


Figure C.6: A double-precision Newton-Raphson iteration-based inverter realised in hardware with only ten NR stages. Top half of the graph shows error for a negative input, bottom half shows error for a positive input.

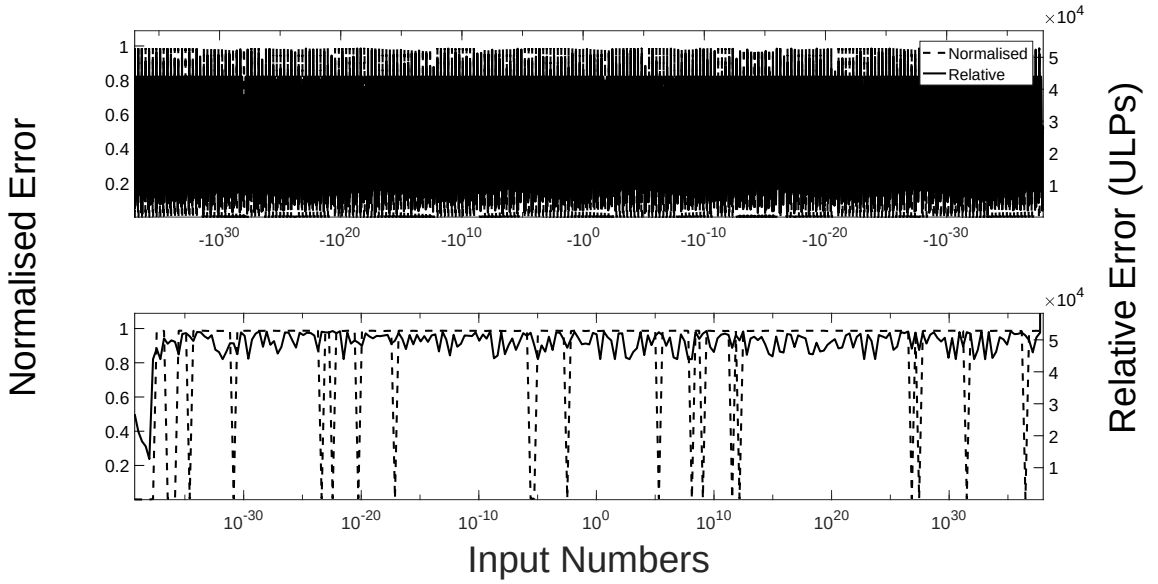


Figure C.7: A single-precision Newton-Raphson iteration-based inverter realised in hardware with only a single NR stage. Top half of the graph shows error for a negative input, bottom half shows error for a positive input.

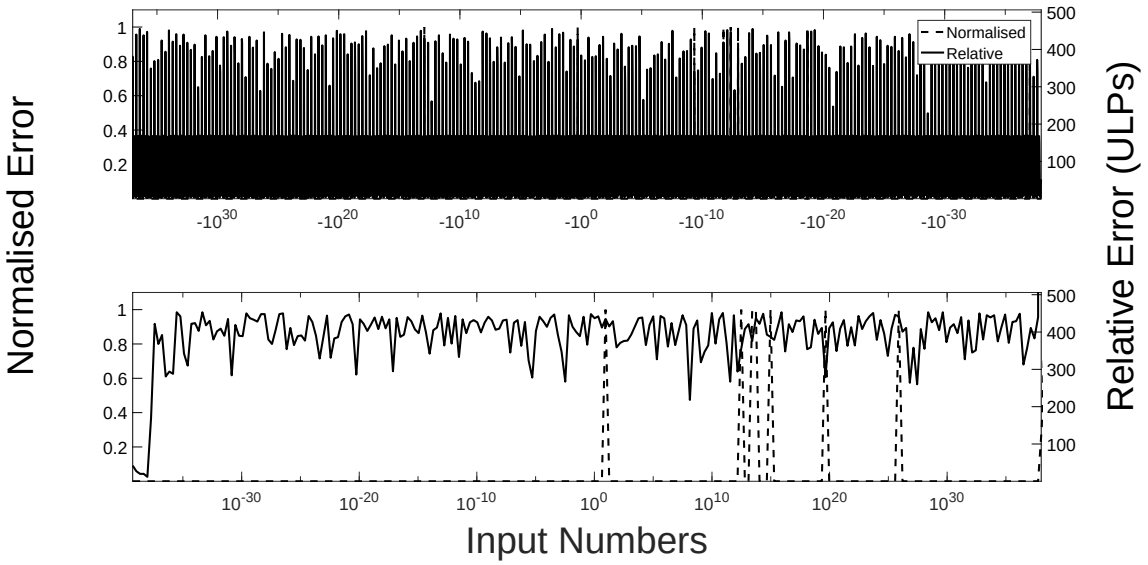


Figure C.8: A single-precision Newton-Raphson iteration-based inverter realised in hardware with only two NR stages. Top half of the graph shows error for a negative input, bottom half shows error for a positive input.

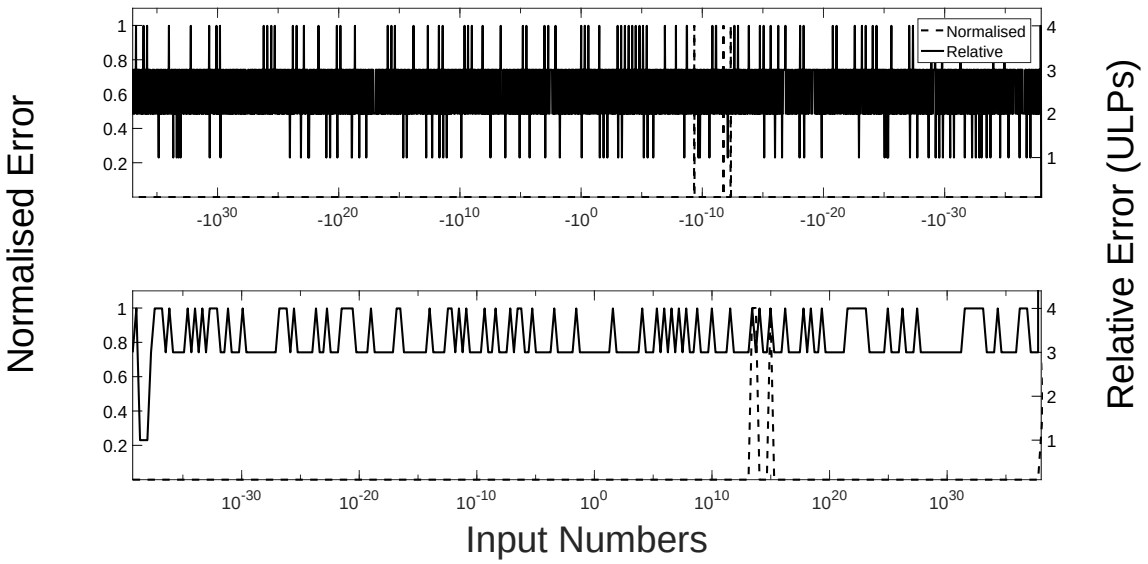


Figure C.9: A single-precision Newton-Raphson iteration-based inverter realised in hardware with only three NR stages. Top half of the graph shows error for a negative input, bottom half shows error for a positive input.

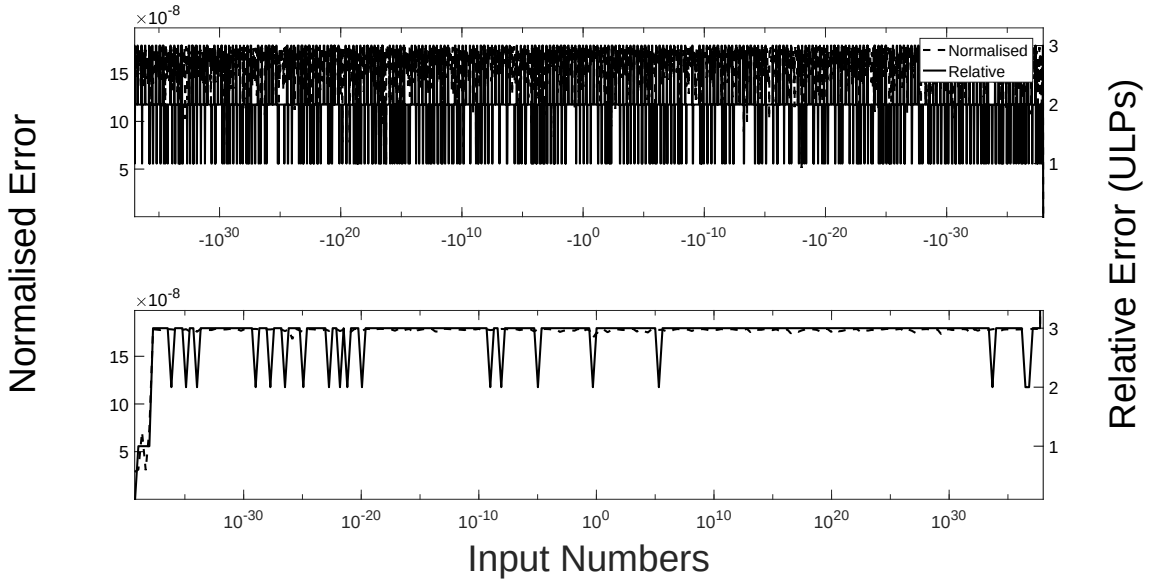


Figure C.10: A single-precision Newton-Raphson iteration-based inverter realised in hardware with only four NR stages. Top half of the graph shows error for a negative input, bottom half shows error for a positive input.

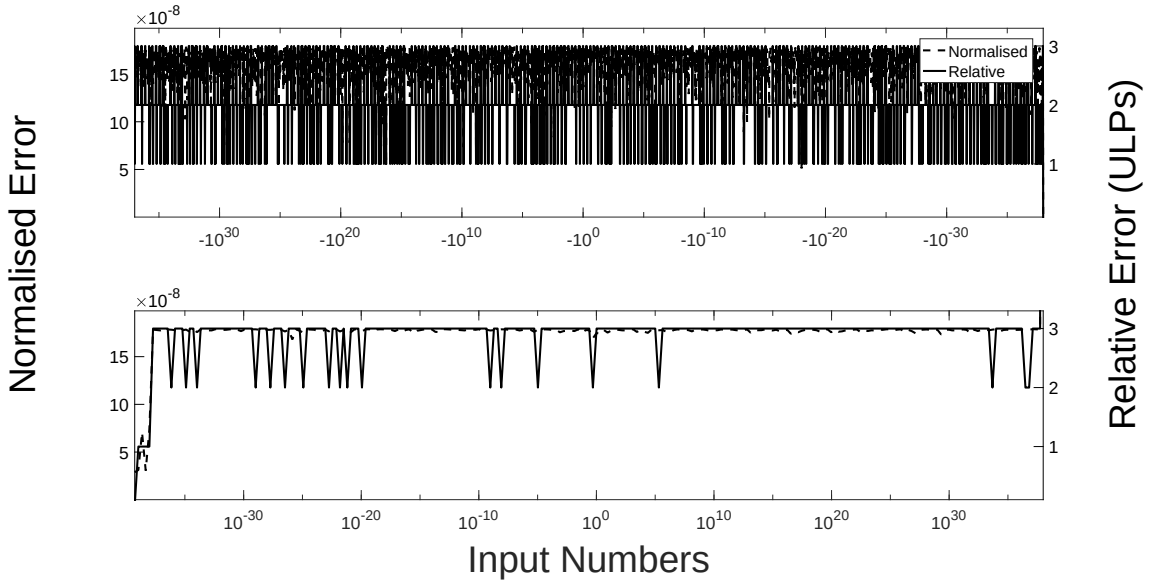


Figure C.11: A single-precision Newton-Raphson iteration-based inverter realised in hardware with only five NR stages. Top half of the graph shows error for a negative input, bottom half shows error for a positive input.

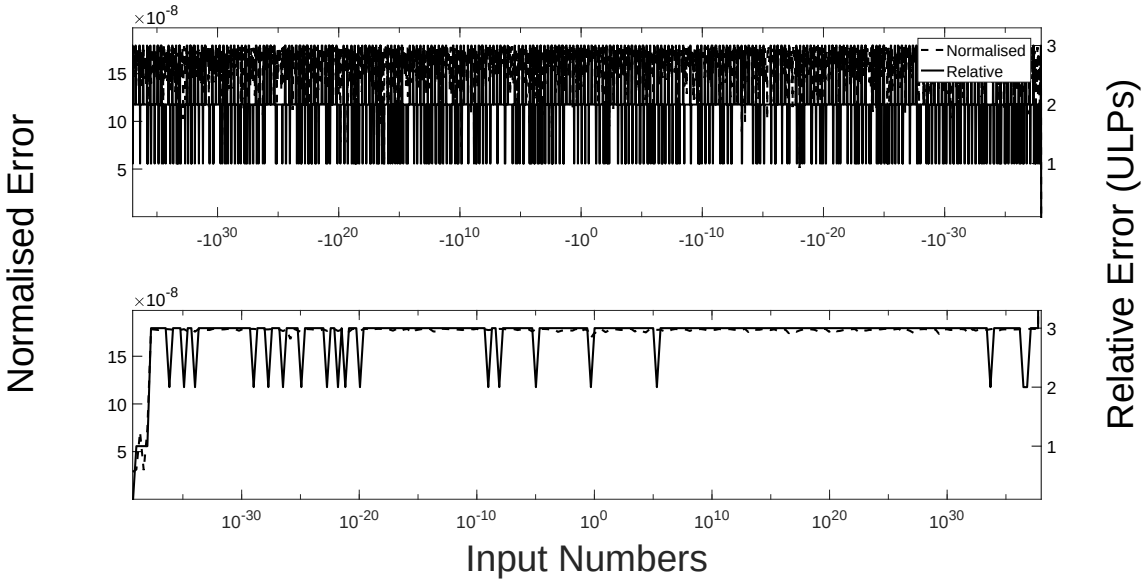


Figure C.12: A single-precision Newton-Raphson iteration-based inverter realised in hardware with only ten NR stages. Top half of the graph shows error for a negative input, bottom half shows error for a positive input.

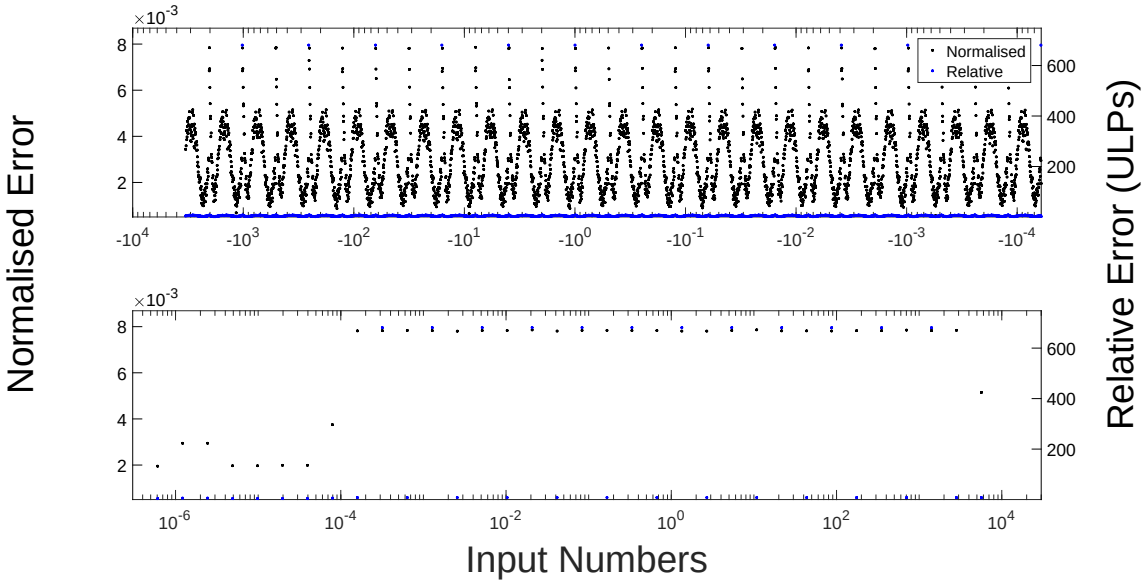


Figure C.13: A half-precision Newton-Raphson iteration-based inverter realised in hardware with only a single NR stage. Top half of the graph shows error for a negative input, bottom half shows error for a positive input. Relative error is marked in blue. Normalise error is marked in black.

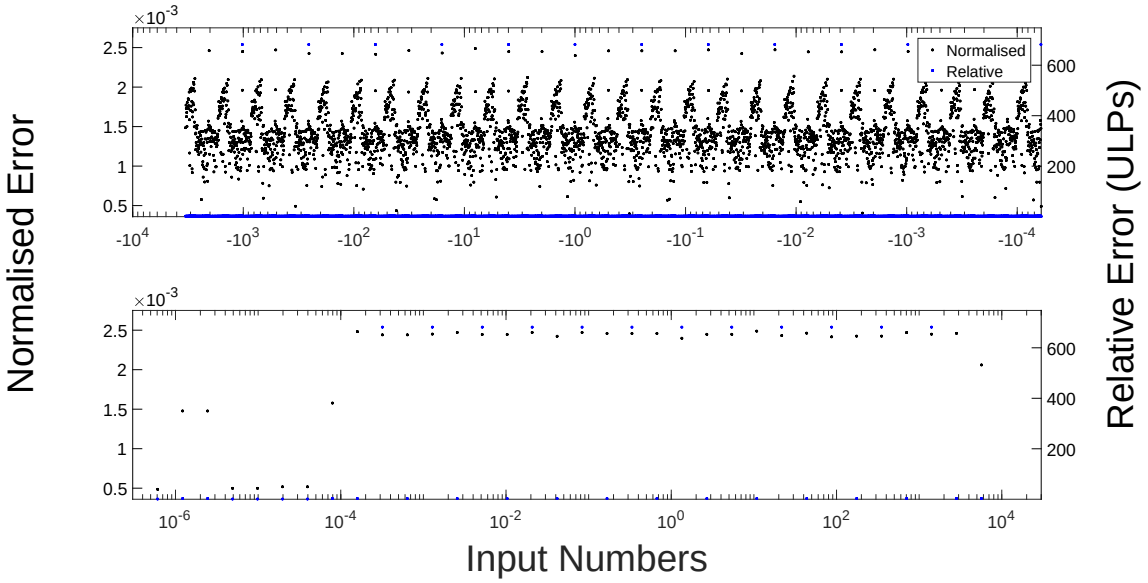


Figure C.14: A half-precision Newton-Raphson iteration-based inverter realised in hardware with only two NR stages. Top half of the graph shows error for a negative input, bottom half shows error for a positive input. Relative error is marked in blue. Normalise error is marked in black.

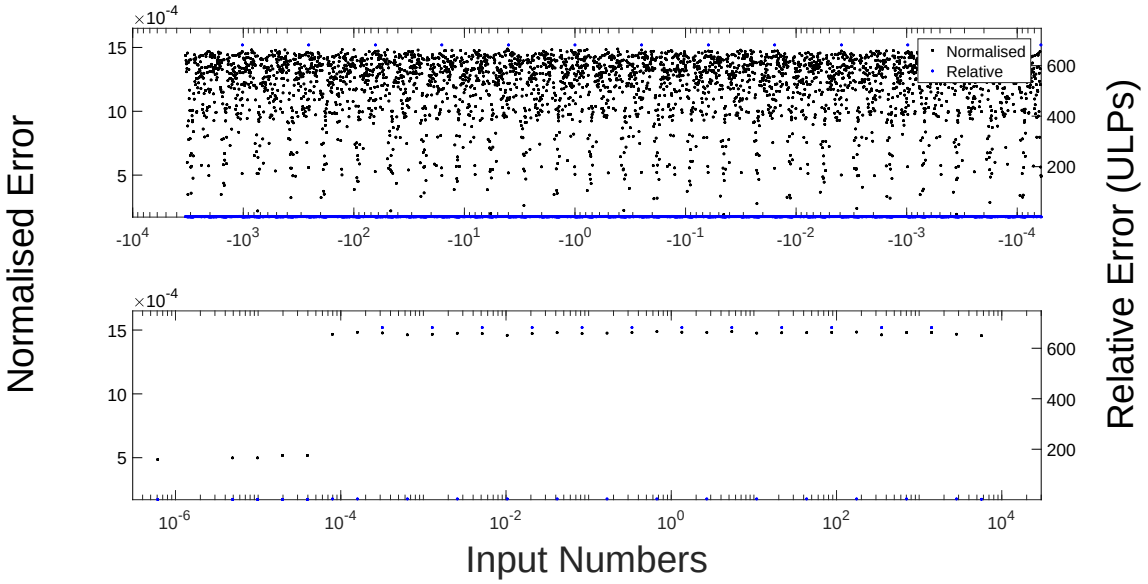


Figure C.15: A half-precision Newton-Raphson iteration-based inverter realised in hardware with only three NR stages. Top half of the graph shows error for a negative input, bottom half shows error for a positive input.

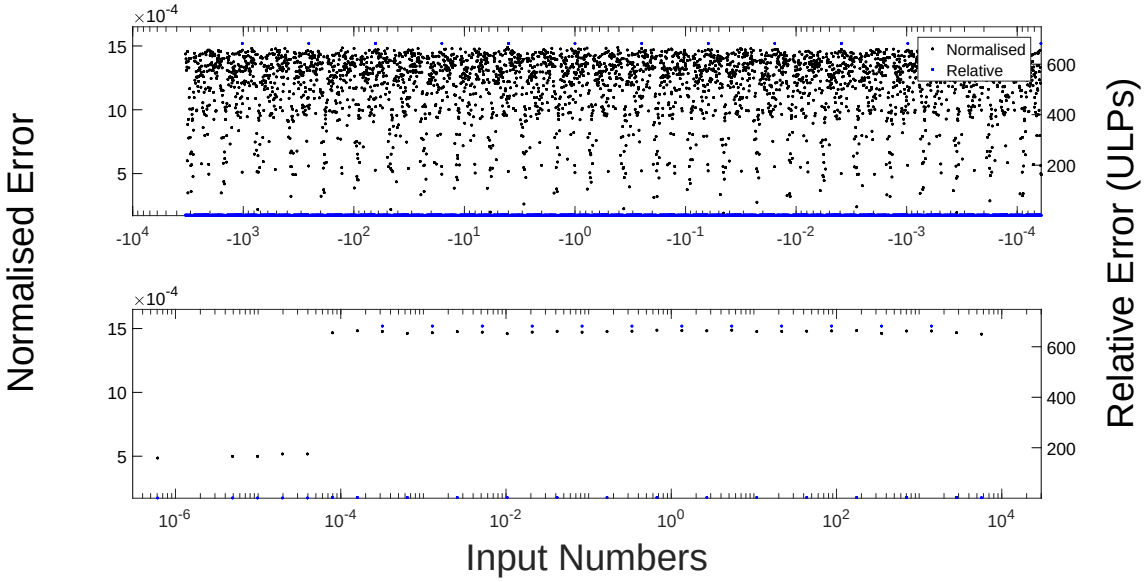


Figure C.16: A half-precision Newton-Raphson iteration-based inverter realised in hardware with only four NR stages. Top half of the graph shows error for a negative input, bottom half shows error for a positive input. Relative error is marked in blue. Normalise error is marked in black.

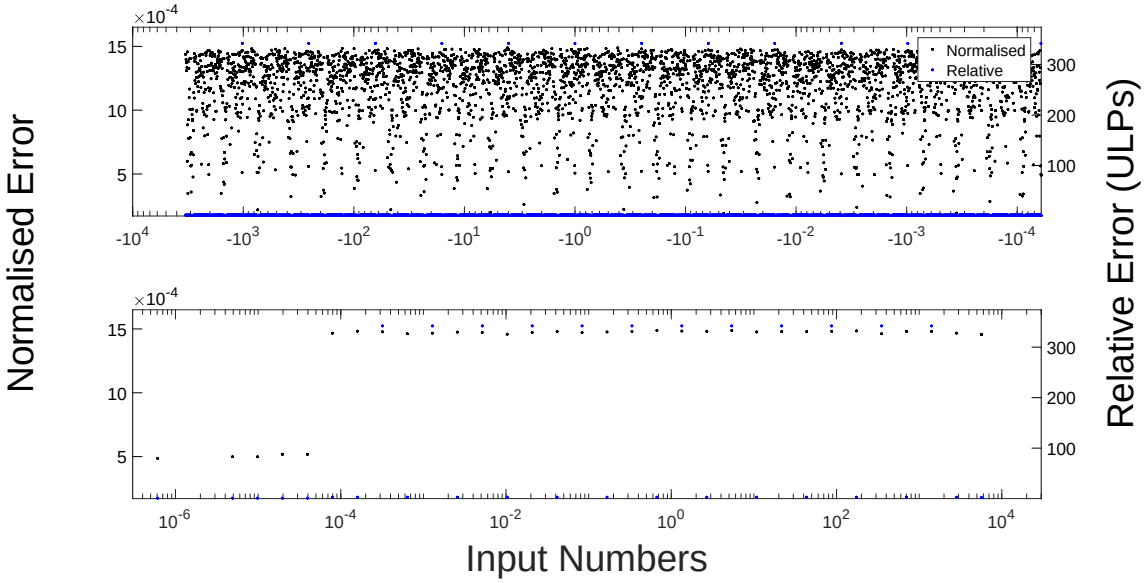


Figure C.17: A half-precision Newton-Raphson iteration-based inverter realised in hardware with only five NR stages. Top half of the graph shows error for a negative input, bottom half shows error for a positive input. Relative error is marked in blue. Normalise error is marked in black.

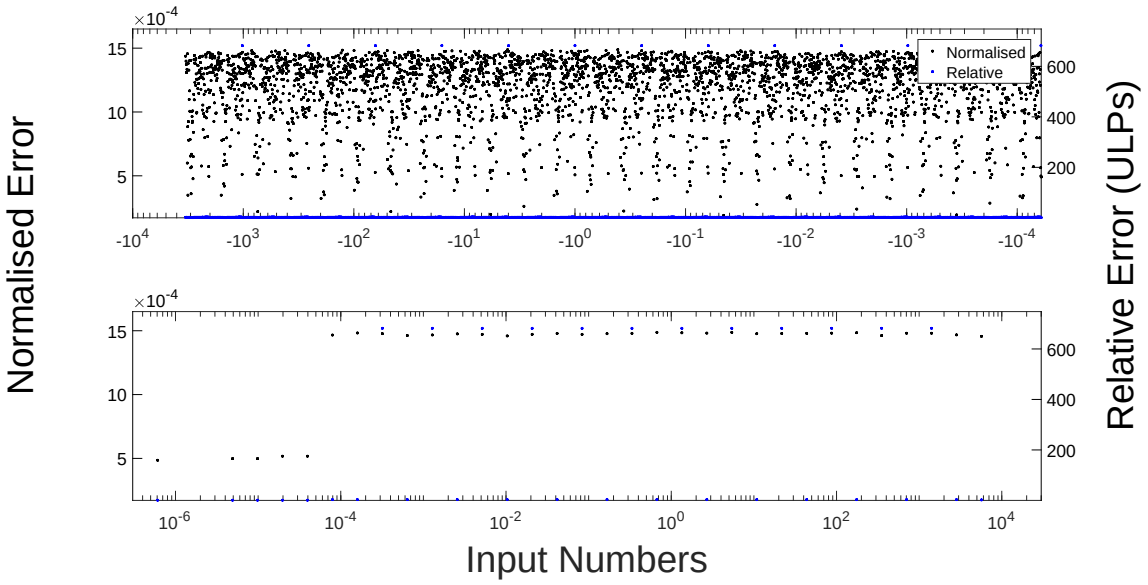


Figure C.18: A half-precision Newton-Raphson iteration-based inverter realised in hardware with only ten NR stages. Top half of the graph shows error for a negative input, bottom half shows error for a positive input. Relative error is marked in blue. Normalise error is marked in black.

Appendix D

Square root accuracy for single and half precision inputs

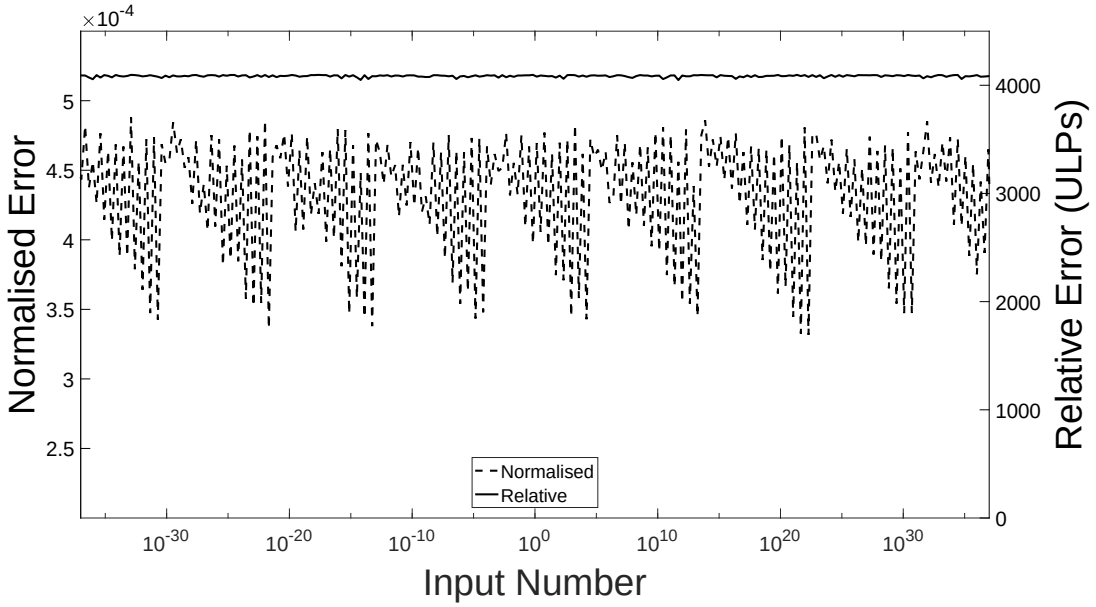


Figure D.1: Relative and absolute error for a traditional non-restoring algorithm in single-precision.

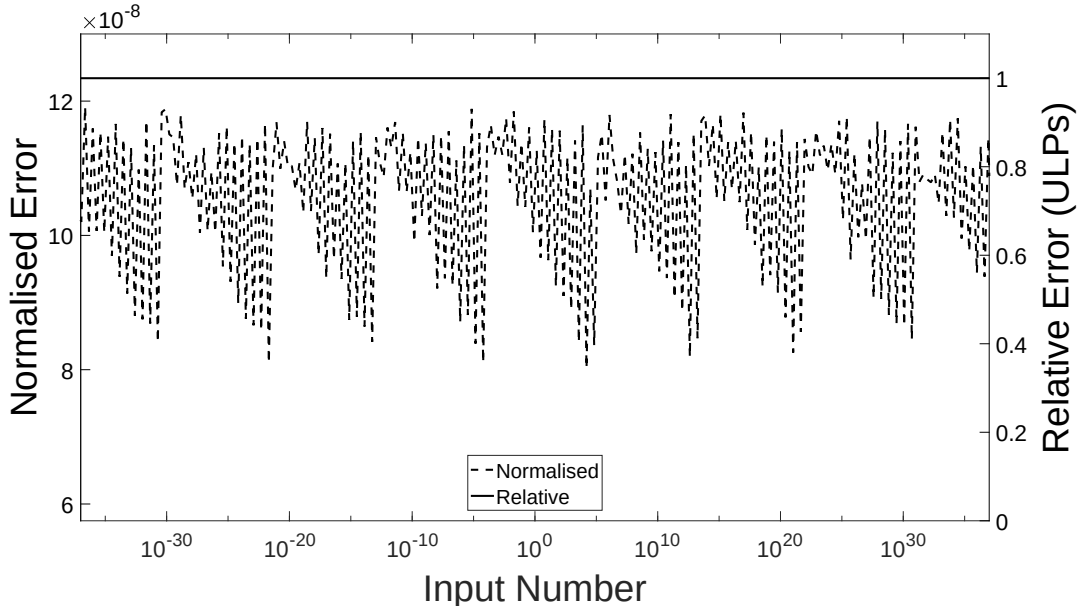


Figure D.2: Relative and absolute error for the new non-restoring algorithm in single-precision.

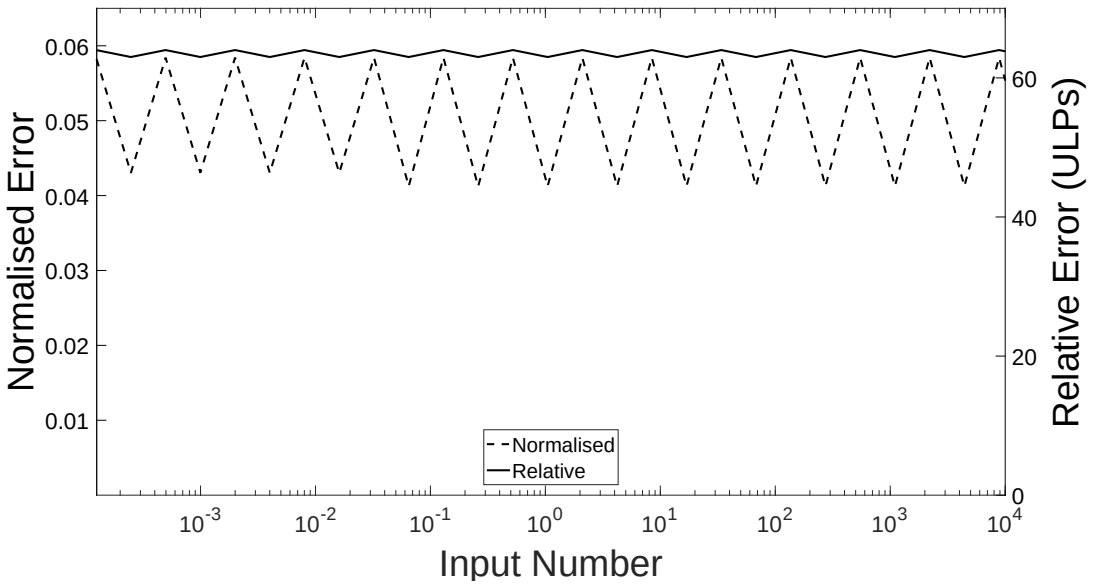


Figure D.3: Relative and absolute error for a traditional non-restoring algorithm in half-precision.

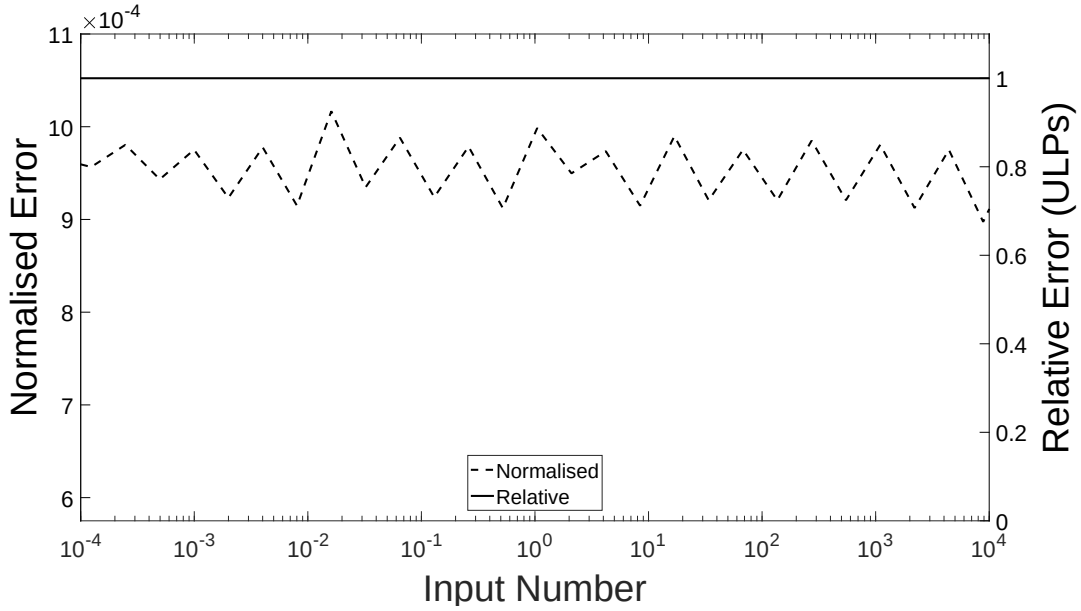


Figure D.4: Relative and absolute error for the new non-restoring algorithm in half-precision.

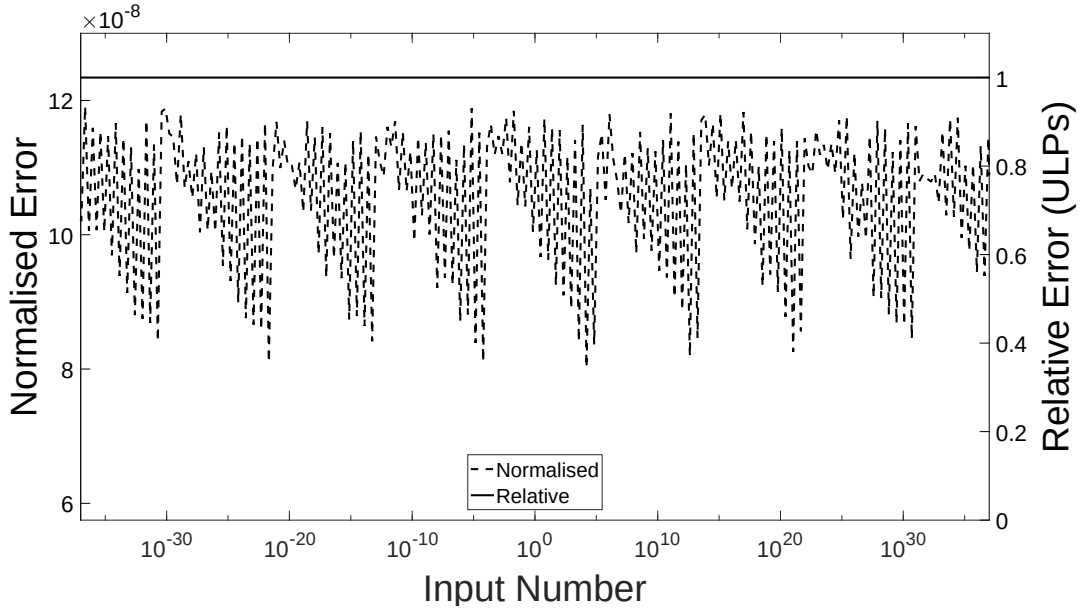


Figure D.5: Adding pipelining to the new method has no impact on the accuracy of the result. Graph shows single-precision.

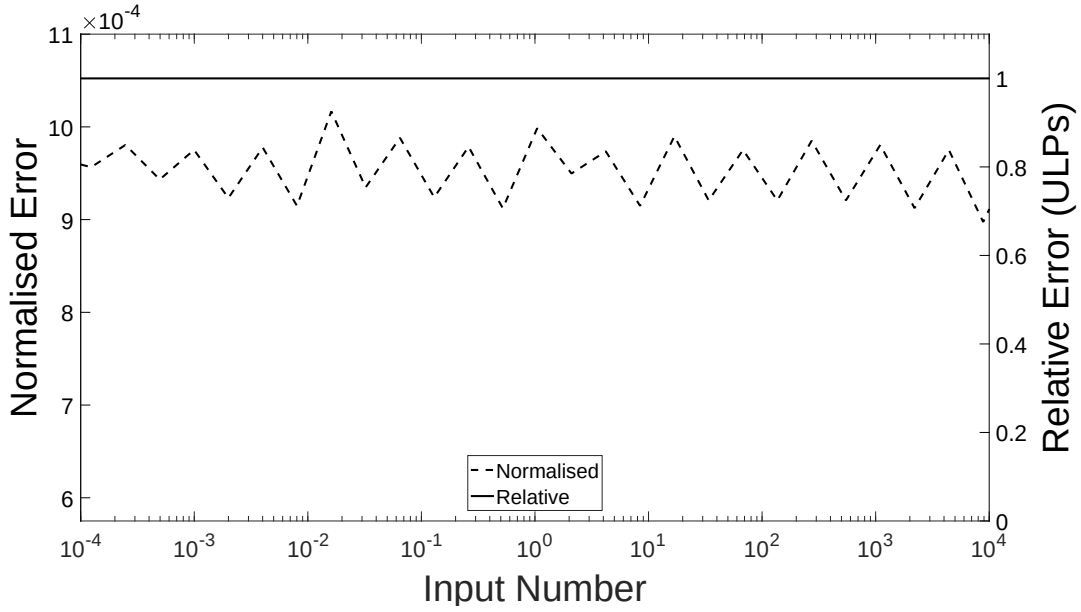


Figure D.6: Adding pipelining to the new method has no impact on the accuracy of the result. Graph shows half-precision.

Appendix E

Approximating $exp(x)$ using both traditional mathematical expansions and hardware friendly interpretations

E.1 Traditional mathematical expansions

E.1.1 Double precision

Table E.1: Resource requirements and timing analysis for Euler and power series mathematical expansions of e^x. Implementations are using double-precision floating-point accuracy, using five stage Newton-Raphson inversion.

Module	Iterations	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs recoverable by Dense Packing	[C] Estimate of ALMs unavailable	Combinational ALUTs	Dedicated logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
euler	1	33222.5 (2194.4)	37591.5 (2320.7)	4492.0 (127.0)	123.0 (0.7)	27293 (4185)	95000 (0)	96	94.82	92.35
power	1	15307.5 (1353.0)	22792.5 (1459.5)	7568.0 (106.8)	83.0 (0.3)	13555 (2090)	41764 (65)	48	96.98	93.74
power	2	30590.0 (2106.4)	35357.0 (2279.6)	4900.0 (174.3)	133.0 (1.1)	25544 (3936)	86803 (65)	96	95.06	92.28

Table E.2: Resource requirements and timing analysis for Euler and power series mathematical expansions of e^x. Implementations are using double-precision floating-point accuracy, using a single stage Newton-Raphson inversion.

Module	Iterations	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs recoverable by Dense Packing	[C] Estimate of ALMs unavailable	Combinational ALUTs	Dedicated logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
euler	1	13799.5 (1313.4)	20799.0 (1424.9)	7062.5 (111.7)	63.0 (0.3)	12342 (2071)	37192 (0)	32	98.26	94.67
euler	2	20734.6 (1845.7)	31373.1 (1981.0)	10764.0 (135.9)	125.5 (0.6)	17022 (2861)	57769 (0)	44	97.1	93.05
euler	3	27344.5 (1930.0)	31808.0 (2078.0)	4566.0 (148.3)	102.5 (0.3)	21818 (3622)	78378 (0)	56	92.81	89.76
euler	4	34218.5 (2335.0)	38249.9 (2460.5)	4176.0 (125.9)	144.5 (0.5)	26455 (4412)	98987 (0)	68	93.08	89.61
power	1	6502.5 (686.2)	9698.0 (748.0)	3222.5 (61.8)	27.0 (0.1)	6215 (1042)	16892 (65)	16	99.46	96.4
power	2	12219.0 (1166.9)	18351.5 (1266.2)	6201.0 (99.4)	68.5 (0.2)	10981 (1817)	32963 (65)	32	94.53	90.48
power	3	18076.0 (1698.2)	26978.0 (1834.0)	9021.0 (136.0)	119.0 (0.3)	15819 (2609)	49071 (65)	48	95.54	93.08
power	4	23842.0 (2191.3)	35512.5 (2324.5)	11849.0 (133.6)	178.5 (0.4)	20459 (3392)	65177 (65)	64	91.7	90.23
power	5	29265.0 (2217.7)	33969.5 (2401.7)	4869.5 (184.6)	165.0 (0.6)	25452 (4205)	81283 (65)	80	95.21	91.73
power	6	35123.5 (2660.6)	38743.0 (2763.6)	3782.5 (104.4)	163.0 (1.3)	30404 (4976)	97391 (65)	96	93.72	91.91

E.1.2 single precision

Table E.3: Resource requirements and timing analysis for Euler and power series mathematical expansions of e^x. Implementations are using single-precision floating-point accuracy, using five stage Newton-Raphson inversion.

Module	Iterations	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs recoverable by Dense Packing	[C] Estimate of ALMs unavailable	Combinational ALUTs	Dedicated logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
euler	1	16660.0 (713.9)	25132.0 (779.0)	8531.0 (65.1)	59.0 (0.1)	13590 (1205)	47928 (0)	24	154.58	149.5
euler	2	26103.5 (1039.9)	38734.0 (1070.7)	12736.0 (30.8)	105.5 (0.0)	19762 (1735)	77063 (0)	35	154.11	148.32
euler	3	35520.0 (1102.6)	39150.0 (1148.8)	3690.5 (46.5)	60.5 (0.3)	26041 (2251)	106294 (0)	46	155.74	149.25
power	1	7595.5 (390.5)	11388.5 (418.8)	3820.5 (28.3)	27.5 (0.0)	6686 (605)	21140 (33)	12	160.0	154.37
power	2	15416.0 (687.0)	23306.5 (757.5)	7967.0 (70.5)	76.5 (0.0)	12795 (1128)	43913 (33)	24	156.57	151.65
power	3	23267.5 (996.9)	34802.5 (1079.3)	11642.0 (82.6)	107.0 (0.2)	18971 (1653)	66787 (33)	36	151.13	144.74
power	4	30991.5 (1083.9)	36237.5 (1186.2)	5322.0 (102.9)	76.0 (0.6)	24978 (2179)	89661 (33)	48	155.79	150.97
power	5	41110.5 (1485.9)	41026.5 (1485.9)	180.5 (0.0)	264.5 (0.0)	31432 (2712)	112533 (33)	60	144.32	140.88

Table E.4: Resource requirements and timing analysis for Euler and power series mathematical expansions of e^x. Implementations are using single-precision floating-point accuracy, using a single stage Newton-Raphson inversion.

Module	Iterations	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs recoverable by Dense Packing	[C] Estimate of ALMs unavailable	Combinational ALUTs	Dedicated logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
euler	1	6841.0 (393.1)	10262.5 (420.0)	3442.5 (26.9)	21.0 (0.0)	6135 (601)	18712 (0)	8	162.76	156.35
euler	2	10197.0 (527.3)	15416.5 (554.3)	5241.0 (27.0)	21.5 (0.0)	8505 (834)	29079 (0)	11	151.93	149.59
euler	3	13671.5 (627.0)	20910.0 (697.1)	7274.0 (70.1)	35.5 (0.0)	10901 (1055)	39478 (0)	14	159.74	153.3
euler	4	17135.0 (771.7)	26261.5 (848.0)	9187.5 (76.2)	61.0 (0.0)	13258 (1284)	49877 (0)	17	155.74	149.54
euler	5	20559.5 (888.8)	31251.5 (970.8)	10804.5 (82.0)	112.5 (0.0)	15648 (1501)	60276 (0)	20	156.69	151.1
euler	6	23982.9 (1024.1)	36177.9 (1096.1)	12283.0 (72.0)	88.0 (0.0)	18041 (1726)	70675 (0)	23	147.91	143.76
euler	7	27358.0 (962.9)	32248.5 (1077.0)	4920.5 (114.1)	30.0 (0.0)	20386 (1951)	81074 (0)	26	157.46	151.95
euler	8	30824.0 (1070.8)	35870.5 (1188.1)	5104.5 (117.5)	58.0 (0.2)	22772 (2176)	91473 (0)	29	150.53	144.18
euler	9	34284.0 (1184.4)	38666.5 (1268.0)	4447.0 (84.2)	64.5 (0.6)	25228 (2401)	101872 (0)	32	148.13	144.7
euler	10	37310.9 (1485.8)	40781.4 (1485.8)	3675.0 (0.0)	204.5 (0.1)	27666 (2626)	112271 (0)	35	151.35	145.14
power	1	3217.0 (209.7)	4718.0 (223.4)	1517.5 (13.7)	16.5 (0.0)	3062 (303)	8516 (33)	4	150.49	144.51
power	2	6067.0 (353.3)	9040.0 (365.7)	3004.0 (12.4)	31.0 (0.0)	5450 (528)	16617 (33)	8	156.59	150.67
power	3	8945.0 (483.4)	13376.5 (516.5)	4490.5 (33.1)	59.0 (0.0)	7934 (760)	24755 (33)	12	155.64	150.08
power	4	11853.5 (599.0)	17666.0 (656.3)	5891.5 (57.3)	79.0 (0.0)	10333 (983)	32893 (33)	16	157.26	152.46
power	5	14774.0 (753.1)	22179.0 (812.7)	7489.5 (59.6)	84.5 (0.0)	12837 (1216)	41029 (33)	20	150.65	144.84
power	6	17659.0 (866.6)	26342.0 (959.8)	8765.0 (93.2)	82.0 (0.0)	15323 (1433)	49167 (33)	24	154.25	148.43
power	7	20452.0 (988.2)	30551.5 (1088.0)	10210.5 (99.8)	111.0 (0.0)	17588 (1653)	57305 (33)	28	154.08	148.02
power	8	23379.0 (1129.2)	34786.0 (1225.8)	11544.5 (96.8)	137.5 (0.3)	20110 (1879)	65441 (33)	32	154.13	148.92
power	9	26272.5 (1262.7)	38053.5 (1321.2)	11899.5 (58.5)	118.5 (0.0)	22550 (2104)	73577 (33)	36	150.38	146.2
power	10	29000.5 (1174.3)	34240.5 (1308.0)	5311.5 (133.6)	71.5 (0.0)	25011 (2337)	81713 (33)	40	152.91	148.77

E.2 Hardware friendly implementations

E.2.1 Double precision

Table E.5: Resource requirements and timing analysis for hardware efficient implementations of approximations of e^x. Implementations are using double precision floating point accuracy.

Module	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs Recoverable by Dense Packing	[C] Estimate of ALMs Unavailable	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
Single quadrant	714.0 (714.0)	723.0 (723.0)	16.5 (16.5)	7.5 (7.5)	1139 (1139)	197 (197)	17	69.94	69.17
Single quadrant with pipeline	702.0 (702.0)	728.5 (728.5)	46.5 (46.5)	20.0 (20.0)	1093 (1093)	390 (390)	17	92.79	90.5
Double quadrant	694.5 (694.5)	702.5 (702.5)	13.5 (13.5)	5.5 (5.5)	1090 (1090)	198 (198)	17	68.06	67.36
Double quadrant with pipeline	704.0 (704.0)	734.0 (734.0)	36.0 (36.0)	6.0 (6.0)	1106 (1106)	397 (397)	17	92.92	89.29
Four quadrant	725.0 (725.0)	727.0 (727.0)	10.0 (10.0)	8.0 (8.0)	1128 (1128)	198 (198)	17	65.87	64.82
Four quadrant with pipeline	722.5 (722.5)	765.0 (765.0)	53.0 (53.0)	10.5 (10.5)	1125 (1125)	421 (421)	17	92.29	89.31
Quadratic fit	764.0 (764.0)	756.0 (756.0)	14.0 (14.0)	22.0 (22.0)	1222 (1222)	197 (197)	25	45.22	44.4
Quadratic fit with pipeline	869.5 (869.5)	922.0 (922.0)	71.0 (71.0)	18.5 (18.5)	1440 (1440)	505 (505)	28	80.48	80.19
Cubic fit	847.0 (847.0)	837.5 (837.5)	17.5 (17.5)	27.0 (27.0)	1377 (1377)	197 (197)	33	32.41	31.51
Cubic fit with pipeline	1167.0 (1167.0)	1240.5 (1240.5)	99.0 (99.0)	25.5 (25.5)	1994 (1994)	747 (747)	42	79.81	79.2
Hybrid using single quadrant	1701.5 (367.5)	2249.5 (389.3)	572.5 (24.5)	24.5 (2.7)	2274 (505)	2864 (131)	17	71.13	70.02
Hybrid using single quadrant with pipeline	1708.0 (370.1)	2196.0 (422.0)	518.5 (52.8)	30.5 (1.0)	2228 (507)	2865 (195)	17	93.55	90.32
Hybrid using double quadrant	1692.0 (361.8)	2203.0 (387.4)	532.5 (25.6)	21.5 (0.0)	2234 (505)	2869 (131)	17	67.67	66.88
Hybrid using double quadrant with pipeline	1681.5 (370.4)	2188.0 (419.3)	523.5 (52.0)	17.0 (3.1)	2255 (506)	2872 (195)	17	92.69	89.74
Hybrid using four quadrant	1684.0 (384.4)	2302.0 (400.3)	640.5 (20.2)	22.5 (4.3)	2268 (509)	2869 (131)	17	65.52	64.54
Hybrid using four quadrant with pipeline	1706.5 (371.2)	2214.5 (429.5)	523.5 (60.5)	15.5 (2.2)	2273 (506)	2896 (195)	17	92.58	88.89
Hybrid using quadratic fit	1745.0 (372.5)	2343.0 (393.5)	609.0 (25.7)	11.0 (4.6)	2361 (505)	2864 (131)	25	43.7	42.8
Hybrid using quadratic fit with pipeline	1817.5 (371.1)	2383.5 (399.2)	583.0 (31.5)	17.0 (3.4)	2570 (505)	2916 (131)	28	86.43	85.99
Hybrid using cubic fit	1854.0 (372.8)	2371.0 (392.6)	545.0 (24.3)	28.0 (4.5)	2515 (505)	2864 (131)	33	32.07	31.41
Hybrid using cubic fit with pipeline	2153.0 (373.0)	2754.5 (395.3)	640.0 (28.1)	38.5 (5.7)	3145 (505)	3288 (131)	42	78.76	77.7
Single quadrant with floating point multiply	639.5 (425.8)	741.5 (452.7)	103.0 (27.8)	1.0 (1.0)	967 (596)	717 (196)	4	99.16	96.22
Two to the power x	426.5 (426.5)	447.0 (447.0)	30.0 (30.0)	9.5 (9.5)	577 (577)	196 (196)	0	171.29	173.58

E.2.2 Single precision

Table E.6: Resource requirements and timing analysis for hardware efficient implementations of approximations of e^x. Implementations are using single precision floating point accuracy.

Module	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs Recoverable by Dense Packing	[C] Estimate of ALMs Unavailable	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
Single quadrant	220.5 (220.5)	228.0 (228.0)	7.5 (7.5)	0.0 (0.0)	345 (345)	121 (121)	4	120.44	119.47
Single quadrant with pipeline	218.0 (218.0)	254.0 (254.0)	36.5 (36.5)	0.5 (0.5)	334 (334)	218 (218)	4	127.44	122.64
Double quadrant	209.5 (209.5)	216.5 (216.5)	7.0 (7.0)	0.0 (0.0)	312 (312)	122 (122)	4	110.77	111.58
Double quadrant with pipeline	218.5 (218.5)	249.0 (249.0)	31.5 (31.5)	1.0 (1.0)	320 (320)	225 (225)	4	127.34	122.55
Four quadrant	218.5 (218.5)	226.5 (226.5)	8.0 (8.0)	0.0 (0.0)	324 (324)	122 (122)	4	110.11	110.28
Four quadrant with pipeline	225.0 (225.0)	258.5 (258.5)	33.5 (33.5)	0.0 (0.0)	341 (341)	244 (244)	4	126.81	122.25
Quadratic fit	206.5 (206.5)	211.5 (211.5)	5.0 (5.0)	0.0 (0.0)	321 (321)	121 (121)	6	87.74	86.9
Quadratic fit with pipeline	235.0 (235.0)	300.5 (300.5)	65.5 (65.5)	0.0 (0.0)	371 (371)	272 (272)	7	127.58	122.76
Cubic fit	207.0 (207.0)	215.5 (215.5)	9.0 (9.0)	0.5 (0.5)	321 (321)	121 (121)	8	64.17	63.07
Cubic fit with pipeline	285.0 (285.0)	351.5 (351.5)	66.5 (66.5)	0.0 (0.0)	458 (458)	359 (359)	11	128.02	122.84
Hybrid using single quadrant	742.0 (130.3)	982.5 (138.4)	242.5 (9.6)	2.0 (1.5)	936 (164)	1454 (67)	4	121.01	118.71
Hybrid using single quadrant with pipeline	726.0 (135.8)	984.0 (155.3)	260.0 (21.3)	2.0 (1.8)	916 (166)	1455 (99)	4	127.49	121.82
Hybrid using double quadrant	733.5 (124.1)	1023.5 (136.7)	291.0 (13.0)	1.0 (0.4)	928 (163)	1459 (67)	4	110.41	111.41
Hybrid using double quadrant with pipeline	735.0 (135.9)	1001.0 (146.3)	270.0 (11.5)	4.0 (1.1)	931 (166)	1462 (99)	4	126.81	121.68
Hybrid using four quadrant	734.5 (128.7)	1009.5 (139.3)	277.0 (12.2)	2.0 (1.5)	936 (166)	1459 (67)	4	109.31	110.17
Hybrid using four quadrant with pipeline	745.5 (127.6)	1012.0 (155.7)	267.0 (28.2)	0.5 (0.1)	947 (166)	1481 (99)	4	126.84	121.65
Hybrid using quadratic fit	715.0 (125.4)	1012.0 (141.2)	297.5 (16.2)	0.5 (0.4)	910 (164)	1454 (67)	6	87.42	85.79
Hybrid using quadratic fit with pipeline	744.5 (127.7)	1002.0 (136.3)	260.5 (10.2)	3.0 (1.5)	952 (164)	1477 (67)	7	125.88	120.93
Hybrid using cubic fit	727.5 (128.1)	990.0 (131.9)	266.0 (5.4)	3.5 (1.6)	910 (164)	1454 (67)	8	63.53	62.48
Hybrid using cubic fit with pipeline	813.0 (127.5)	1119.5 (143.4)	307.0 (15.9)	0.5 (0.0)	1073 (164)	1630 (67)	11	125.08	120.05
Single quadrant with floating point multiply	279.0 (188.5)	339.0 (197.3)	60.0 (8.8)	0.0 (0.0)	454 (289)	365 (100)	1	175.25	174.58
Two to the power x	189.0 (189.0)	198.5 (198.5)	9.5 (9.5)	0.0 (0.0)	288 (288)	100 (100)	0	219.83	224.42

E.2.3 Error plots for the hardware friendly $exp(x)$ implementations

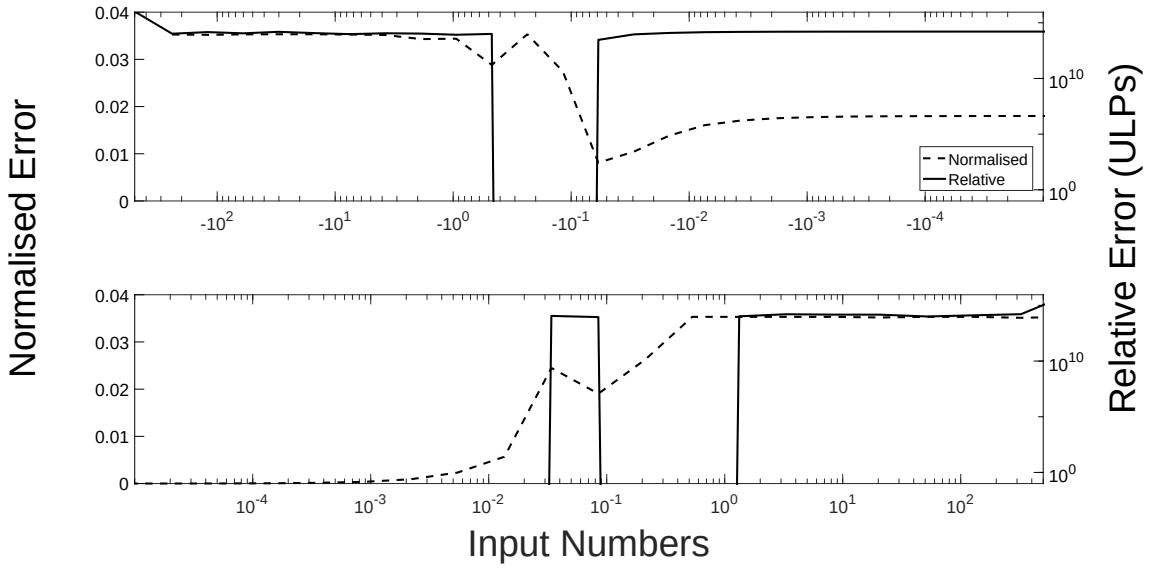


Figure E.1: Hardware friendly floating-point exponent approximation using a single line curve fit in double-precision.

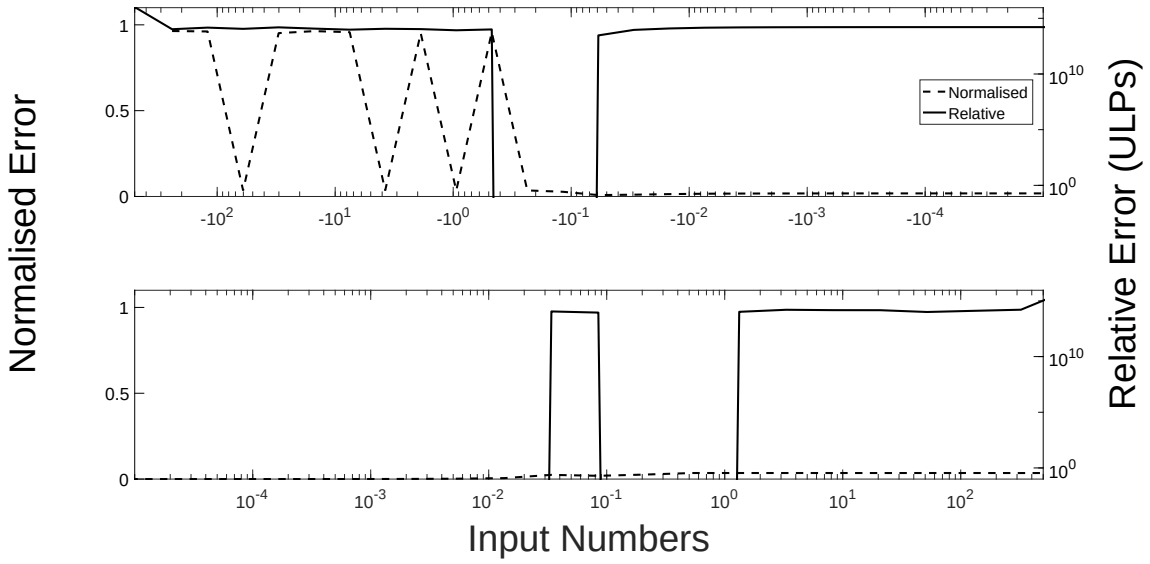


Figure E.2: Hardware friendly floating-point exponent approximation using a single line curve fit with integer divide in double-precision.

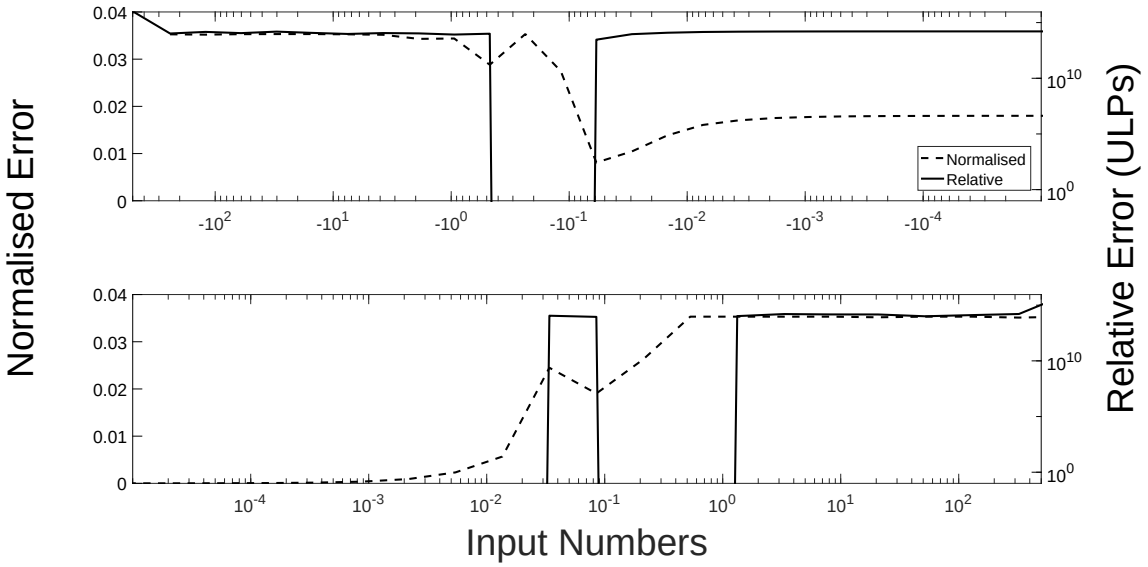


Figure E.3: Hardware friendly floating-point exponent approximation using a single line curve with floating-point multiply in double-precision.

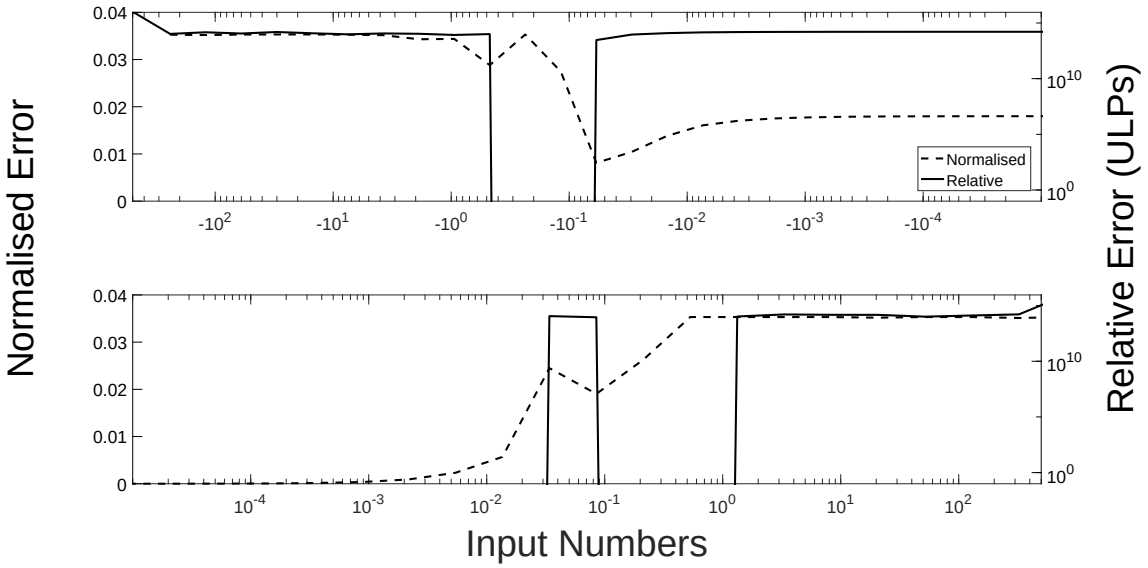


Figure E.4: Hardware friendly floating-point exponent approximation using a single line curve fit with pipelining in double-precision.

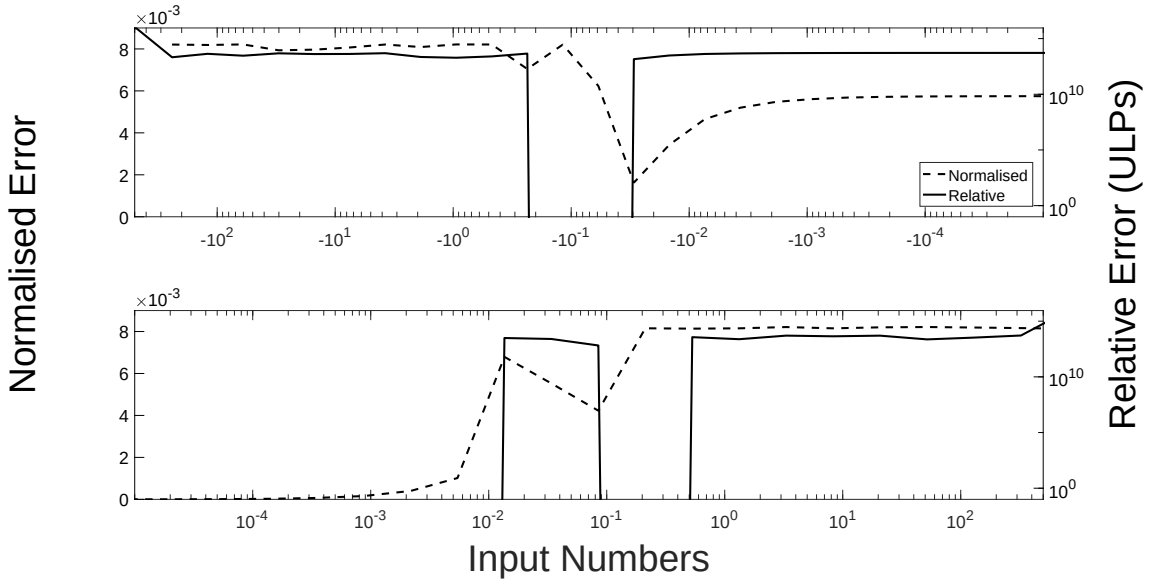


Figure E.5: Hardware friendly floating-point exponent approximation using a double line curve fit in double-precision.

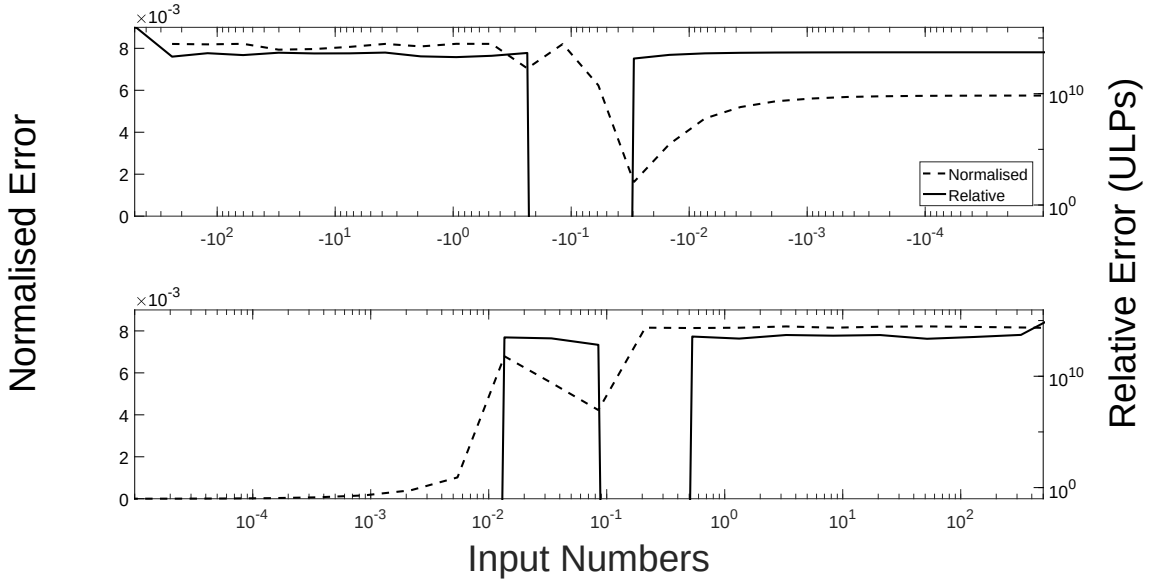


Figure E.6: Hardware friendly floating-point exponent approximation using a double line curve fit with pipelining in double-precision.

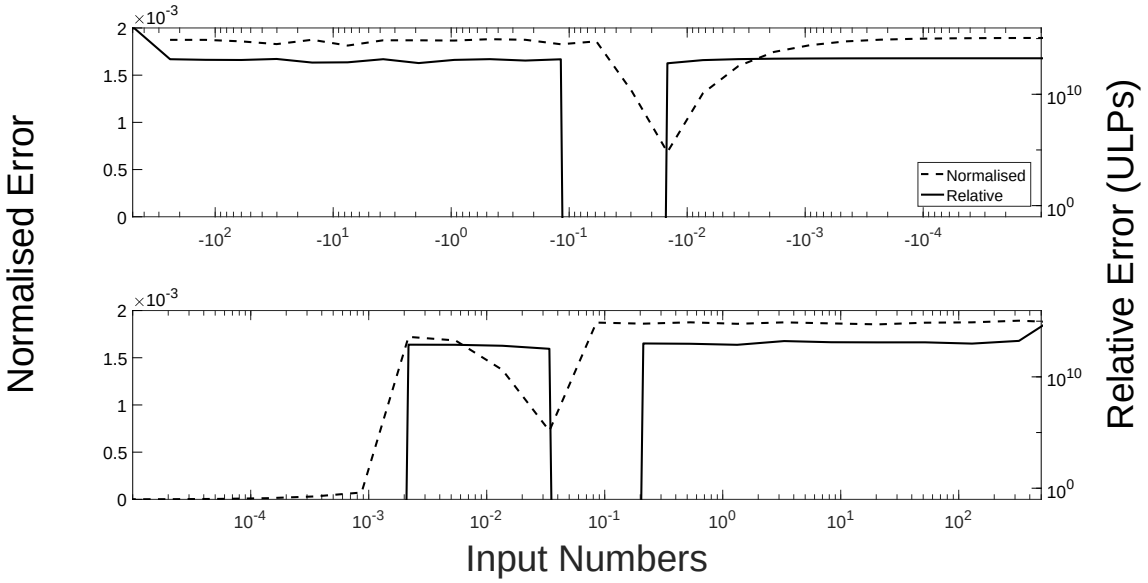


Figure E.7: Hardware friendly floating-point exponent approximation using a four line curve fit in double-precision.

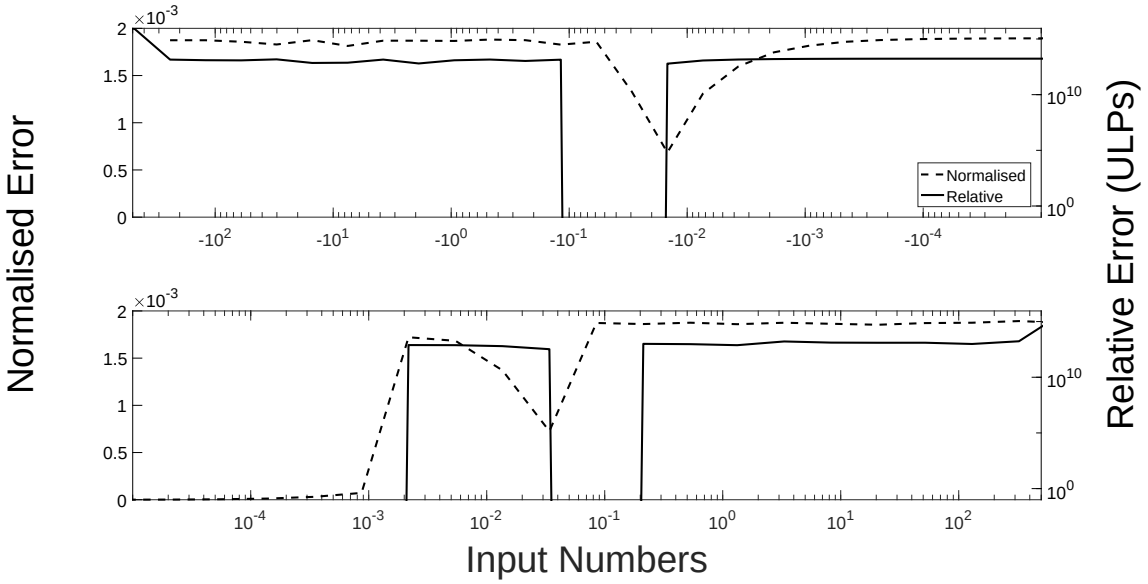


Figure E.8: Hardware friendly floating-point exponent approximation using a four line curve fit with pipelining in double-precision.

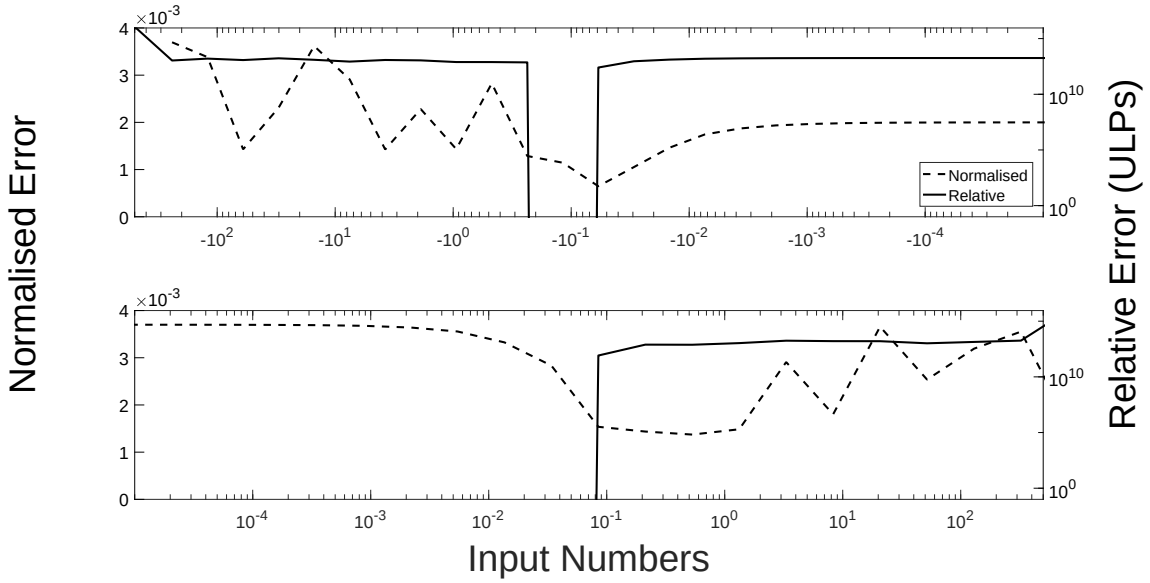


Figure E.9: Hardware friendly floating-point exponent approximation using a quadratic curve fit in double-precision.

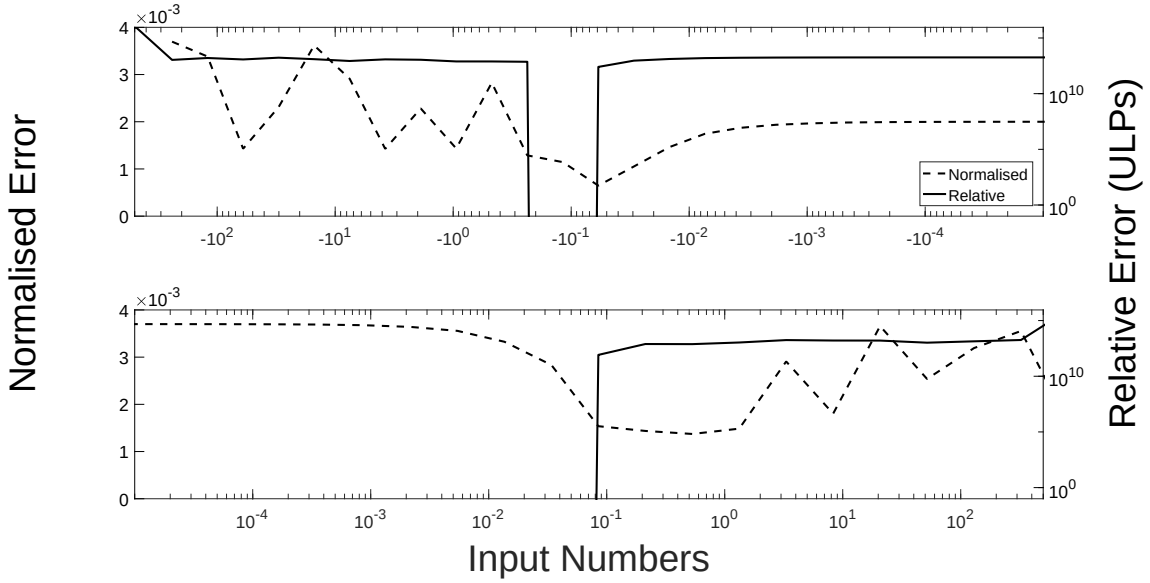


Figure E.10: Hardware friendly floating-point exponent approximation using a quadratic curve fit with pipelining in double-precision.

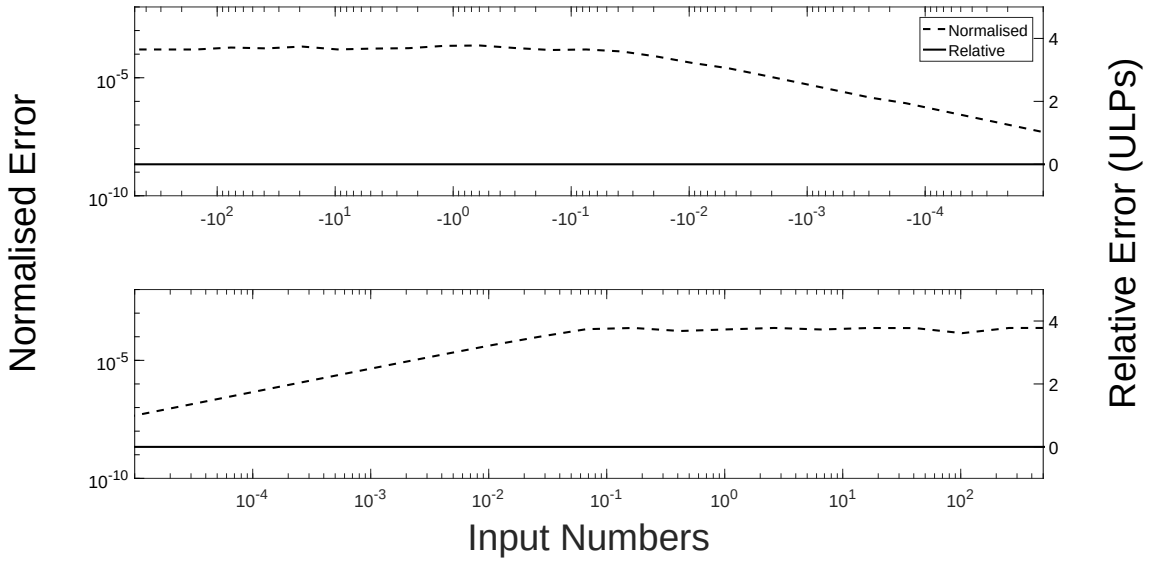


Figure E.11: Hardware friendly floating-point exponent approximation using a cubic curve fit in double-precision.

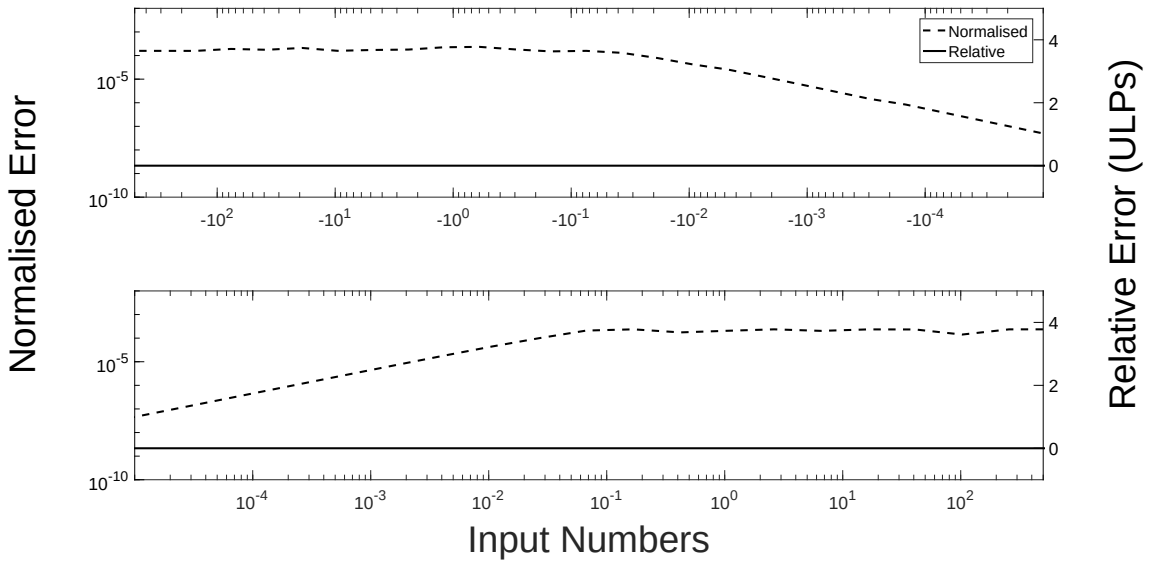


Figure E.12: Hardware friendly floating-point exponent approximation using a cubic curve fit with pipelining in double-precision.

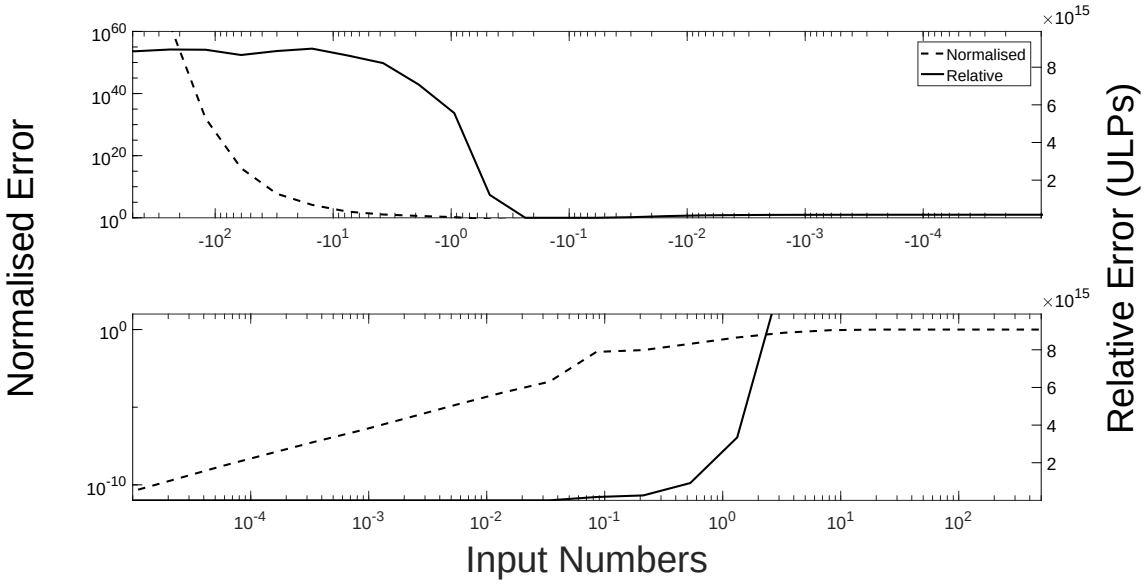


Figure E.13: Hardware floating-point two to the x approximation in double-precision.

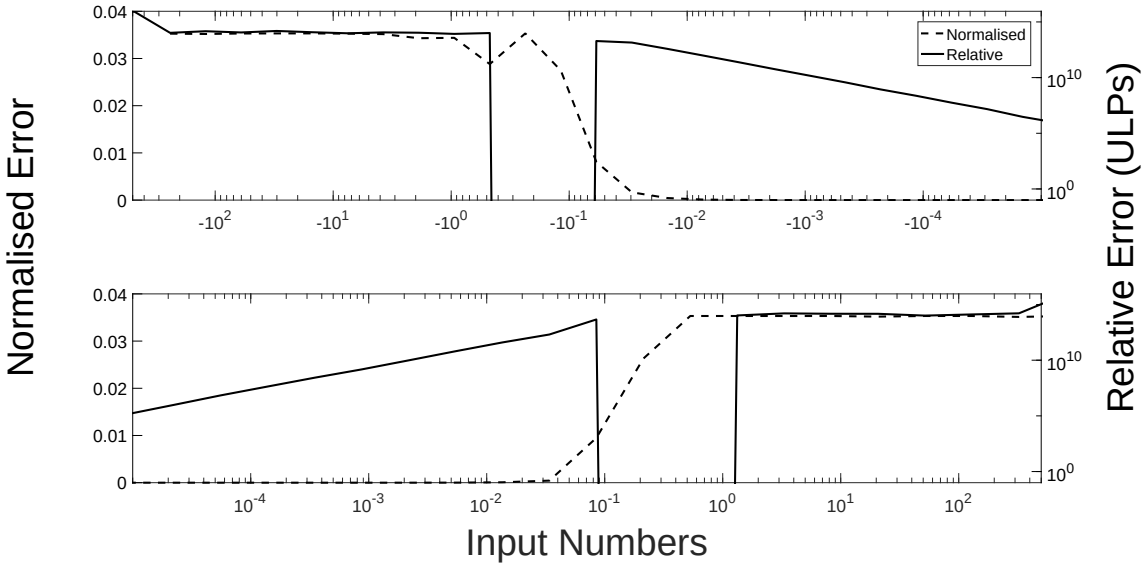


Figure E.14: Hardware friendly floating-point exponent hybrid approximation single line curve fit and $1 + x$ in double-precision.

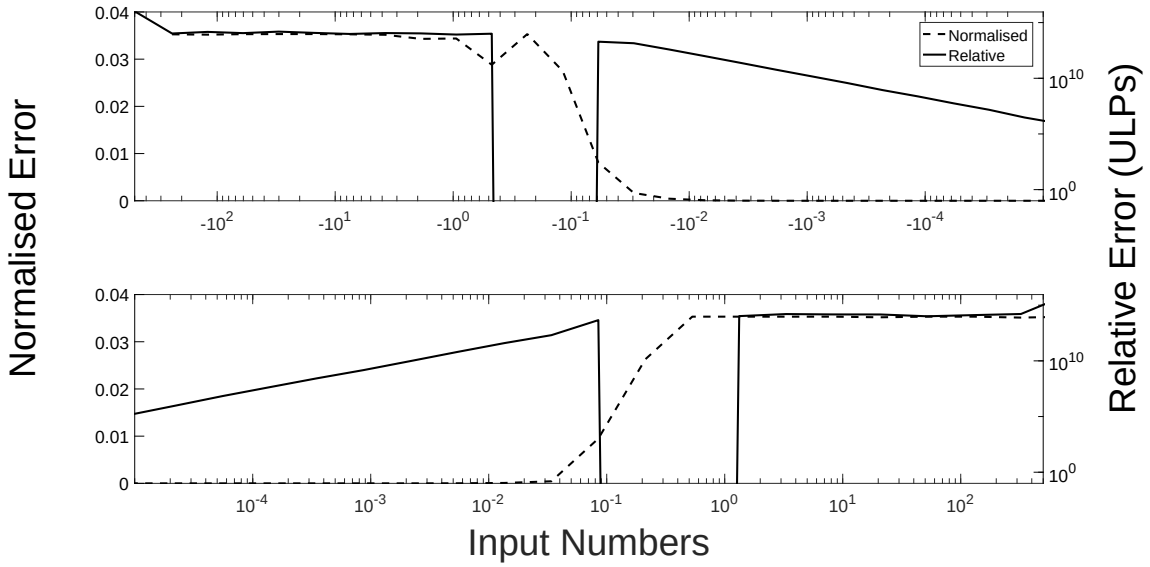


Figure E.15: Hardware friendly floating-point exponent hybrid approximation single line curve fit and $1 + x$ with pipelining in double-precision.

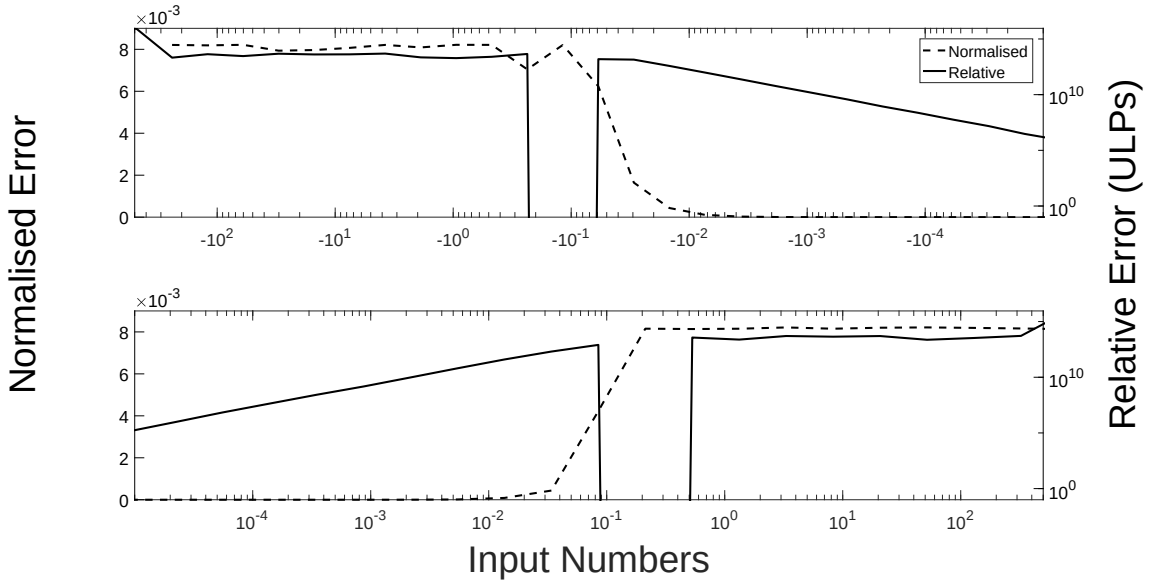


Figure E.16: Hardware friendly floating-point exponent hybrid approximation double line curve fit and $1 + x$ in double-precision.

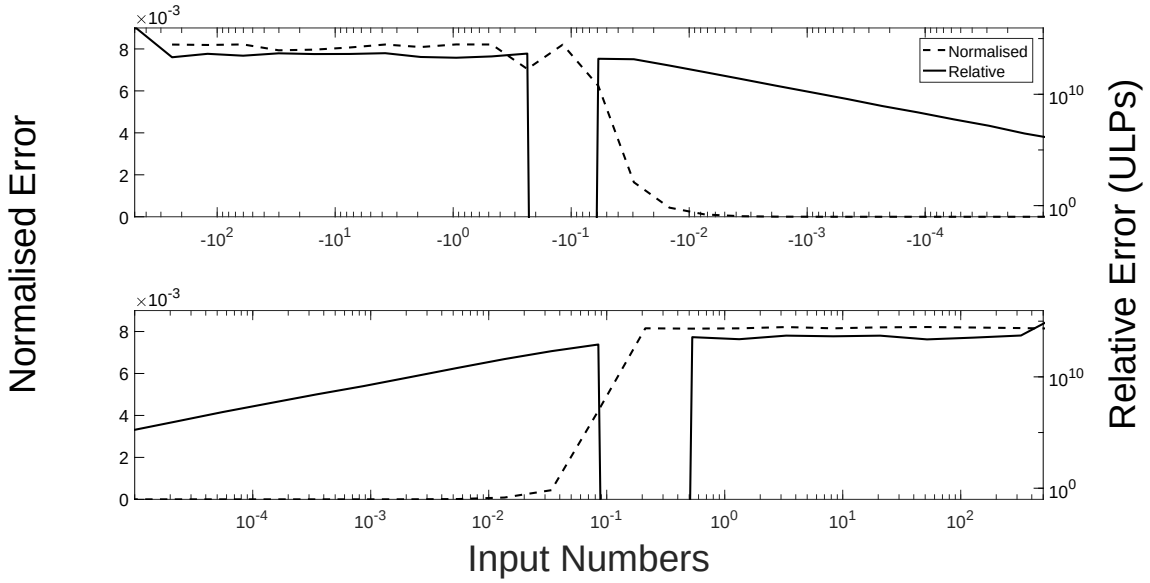


Figure E.17: Hardware friendly floating-point exponent hybrid approximation double line curve fit and $1 + x$ with pipelining in double-precision.

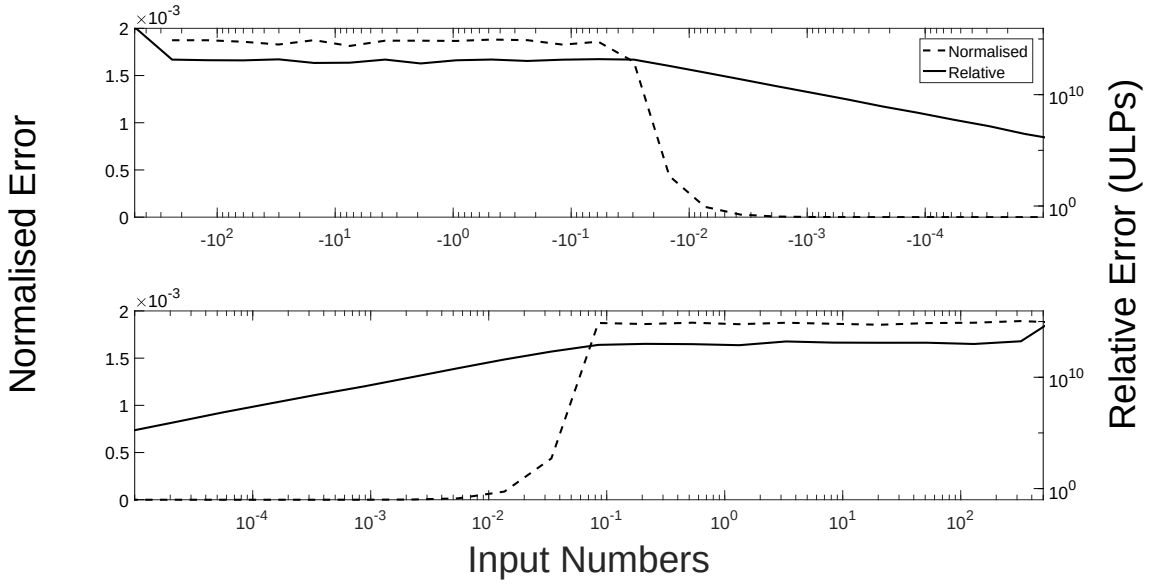


Figure E.18: Hardware friendly floating-point exponent hybrid approximation four line curve fit and $1 + x$ in double-precision.

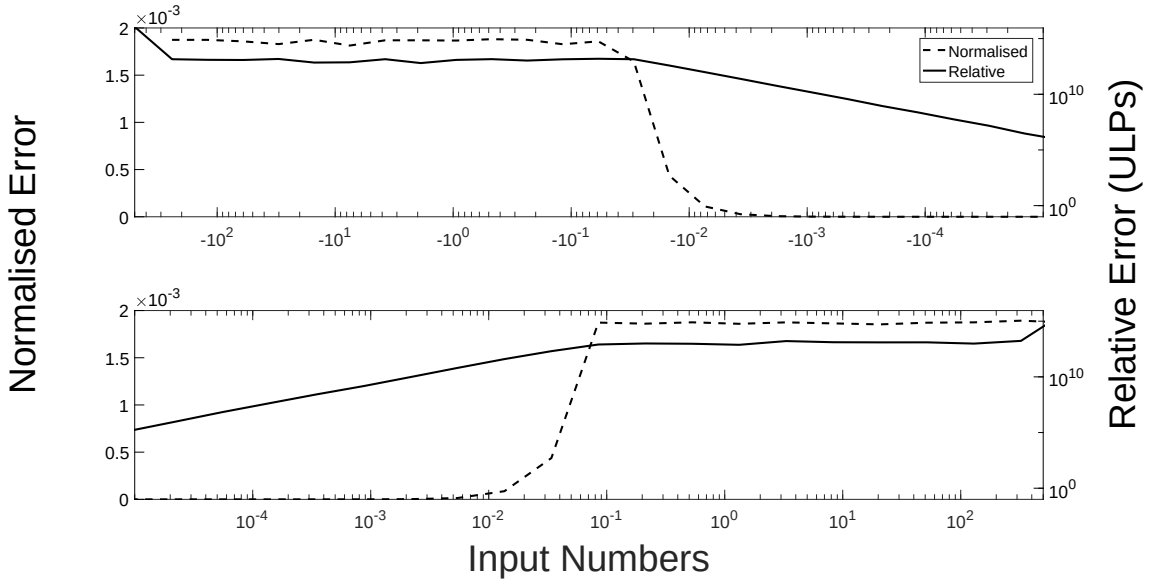


Figure E.19: Hardware friendly floating-point exponent hybrid approximation four line curve fit and $1 + x$ with pipelining in double-precision.

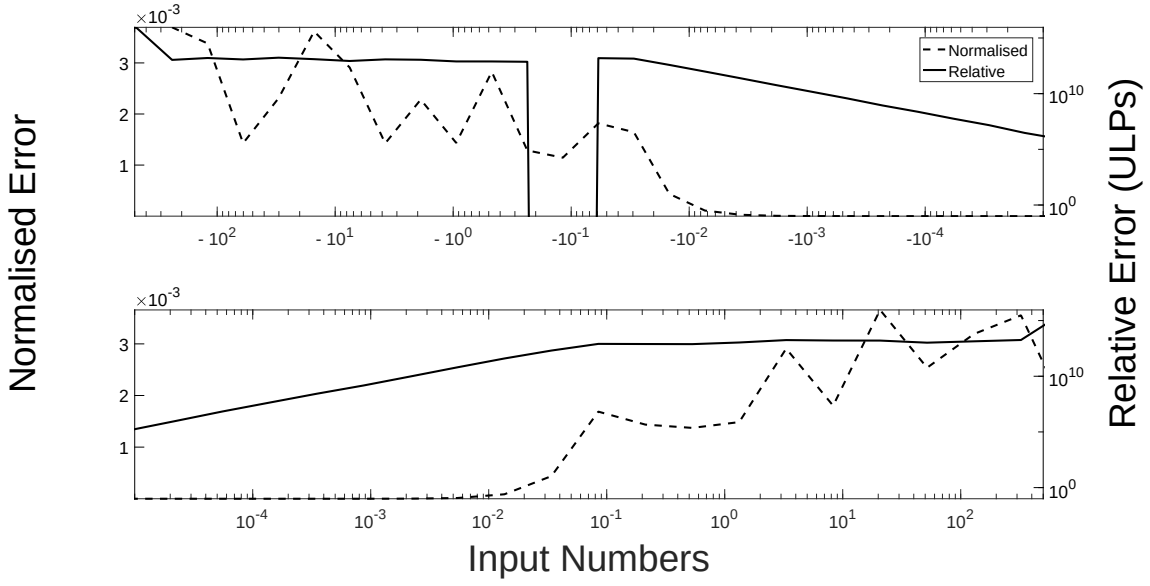


Figure E.20: Hardware friendly floating-point exponent hybrid approximation quadratic curve fit and $1 + x$ in double-precision.

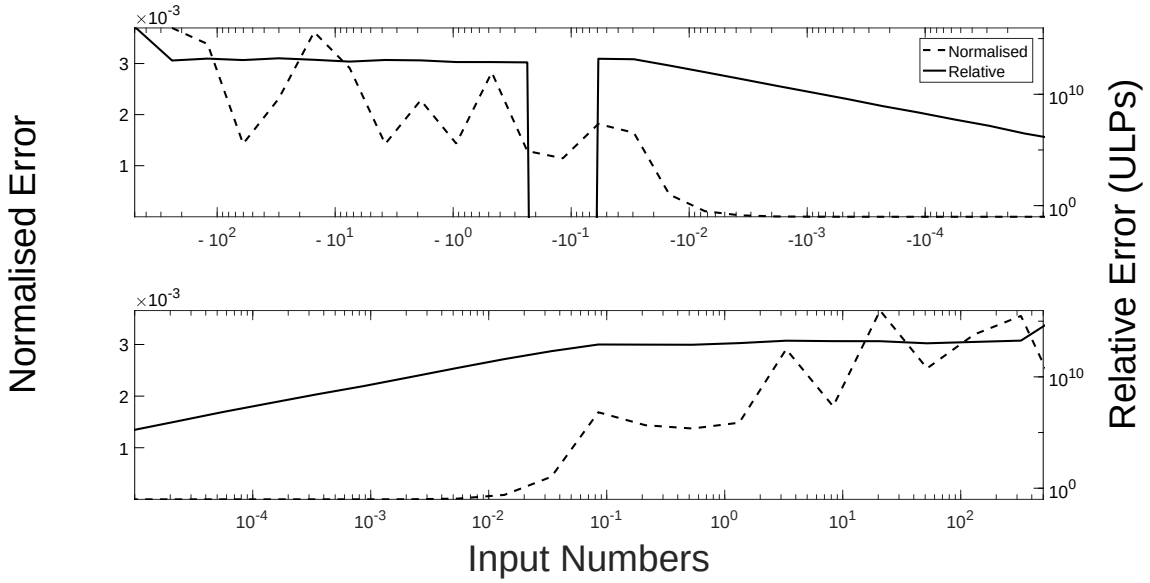


Figure E.21: Hardware friendly floating-point exponent hybrid approximation quadratic curve fit and $1 + x$ with pipelining in double-precision.

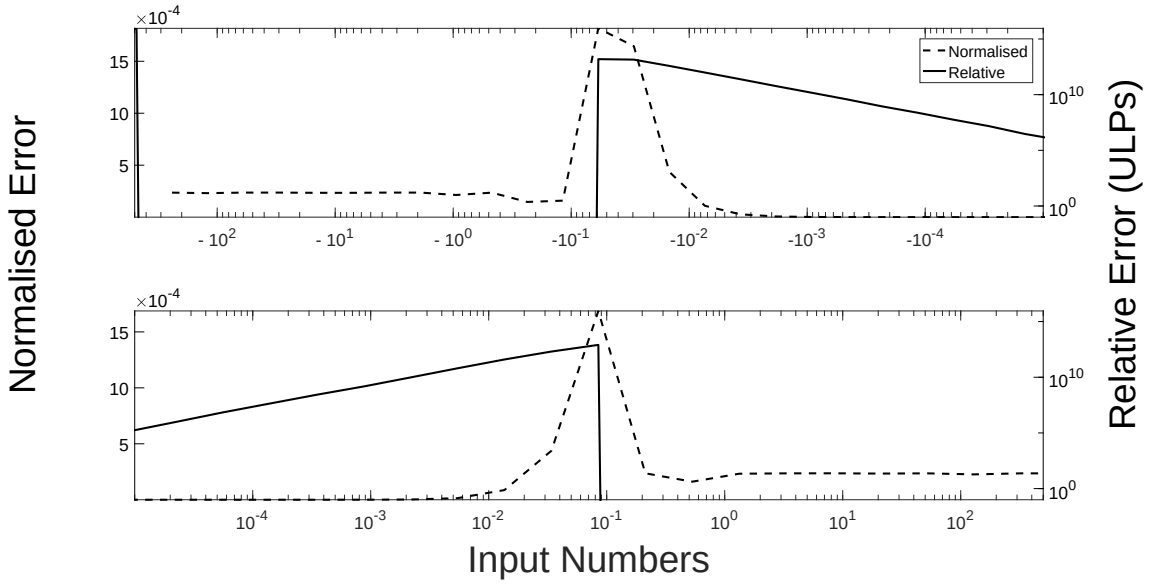


Figure E.22: Hardware friendly floating-point exponent hybrid approximation cubic curve fit and $1 + x$ in double-precision.

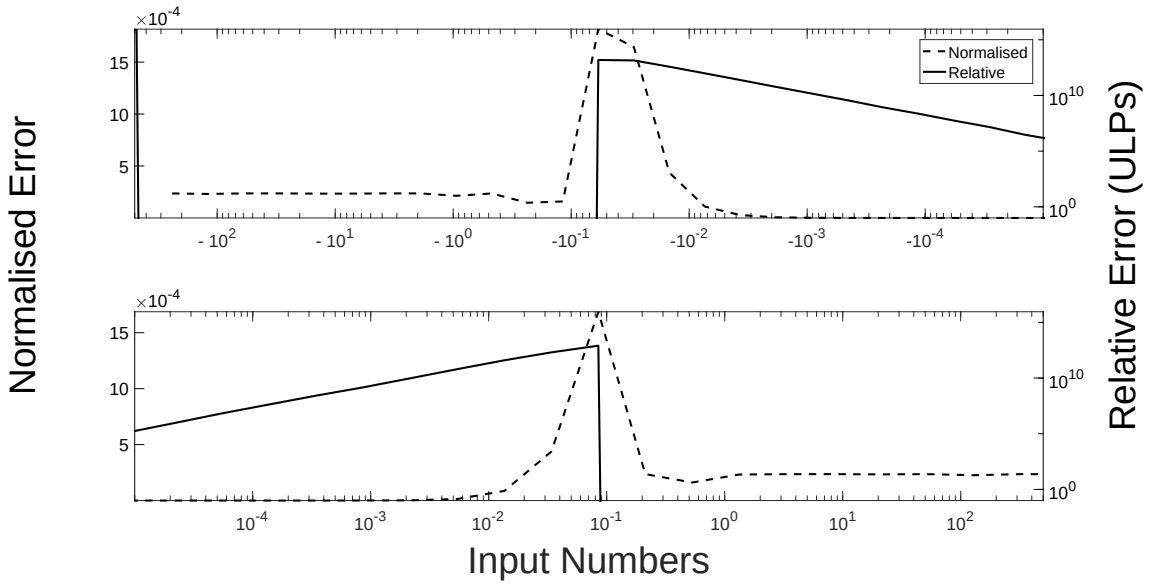


Figure E.23: Hardware friendly floating-point exponent hybrid approximation cubic curve fit and $1 + x$ with pipelining in double-precision.

Appendix F

Hardware Implementations of the Hodgkin-Huxley Model of a Neuron

F.1 Step responses

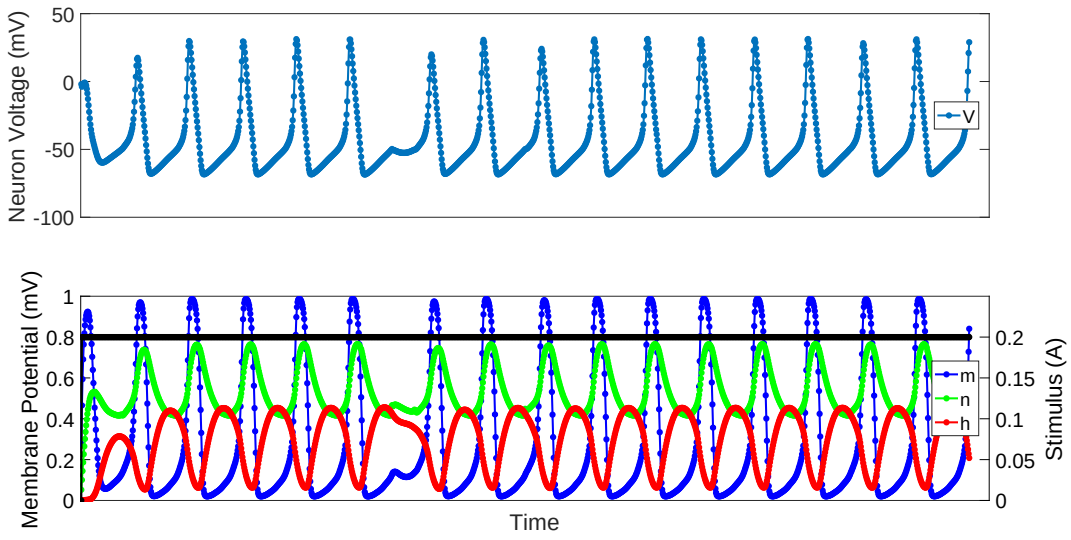


Figure F.1: Hardware implementation of a Hodgkin-Huxley neuron using single line piecewise linear approximation for the exponential function. The neurons response to a step input of 0.2 A (black trace, bottom graph). m, n, and h are the membrane potentials internal to the neuron.

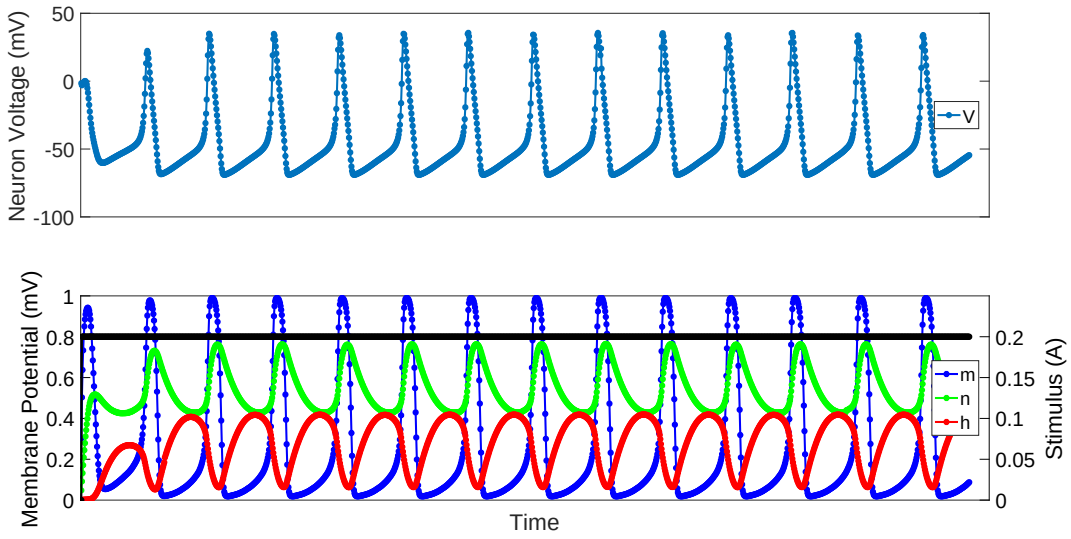


Figure F.2: Hardware implementation of a Hodgkin-Huxley neuron using two line piecewise linear approximation for the exponential function. The neurons response to a step input of 0.2 A (black trace, bottom graph). m, n, and h are the membrane potentials internal to the neuron.

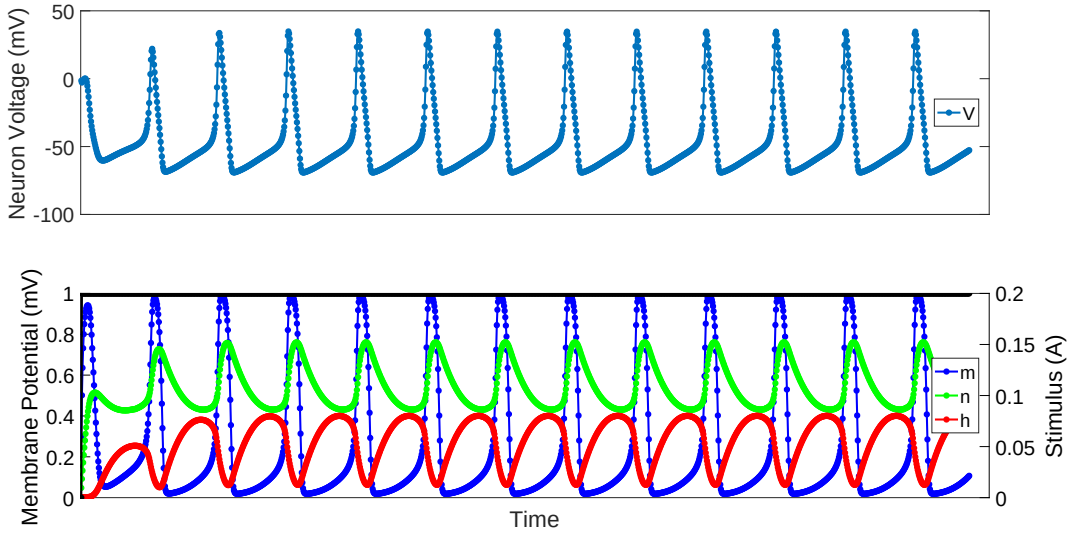


Figure F.3: Hardware implementation of a Hodgkin-Huxley neuron using four line piecewise linear approximation for the exponential function. The neurons response to a step input of 0.2 A (black trace, bottom graph). m, n, and h are the membrane potentials internal to the neuron.

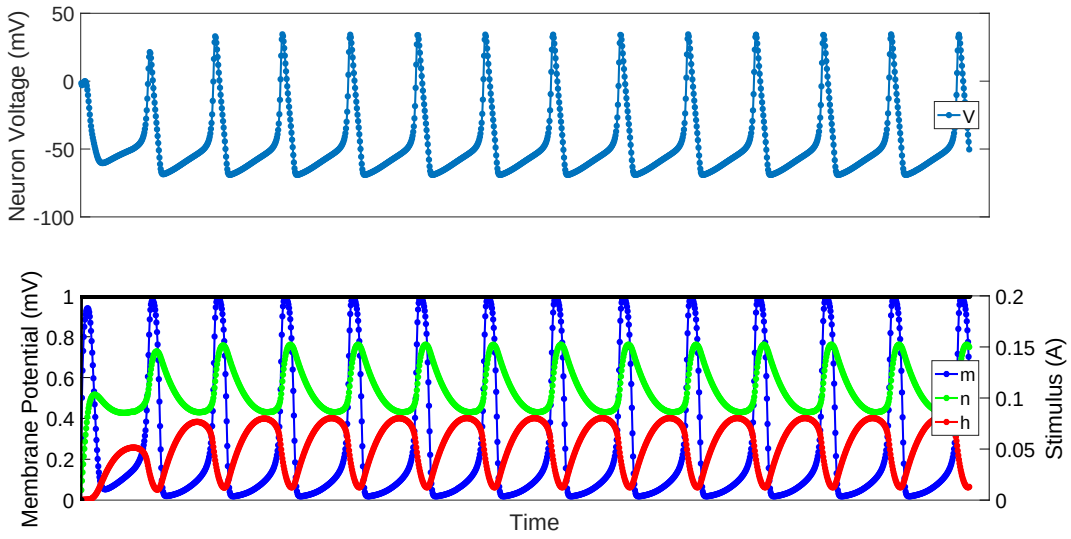


Figure F.4: Hardware implementation of a Hodgkin-Huxley neuron using quadratic curve fitting approximation for the exponential function. The neurons response to a step input of 0.2 A (black trace, bottom graph). m, n, and h are the membrane potentials internal to the neuron.

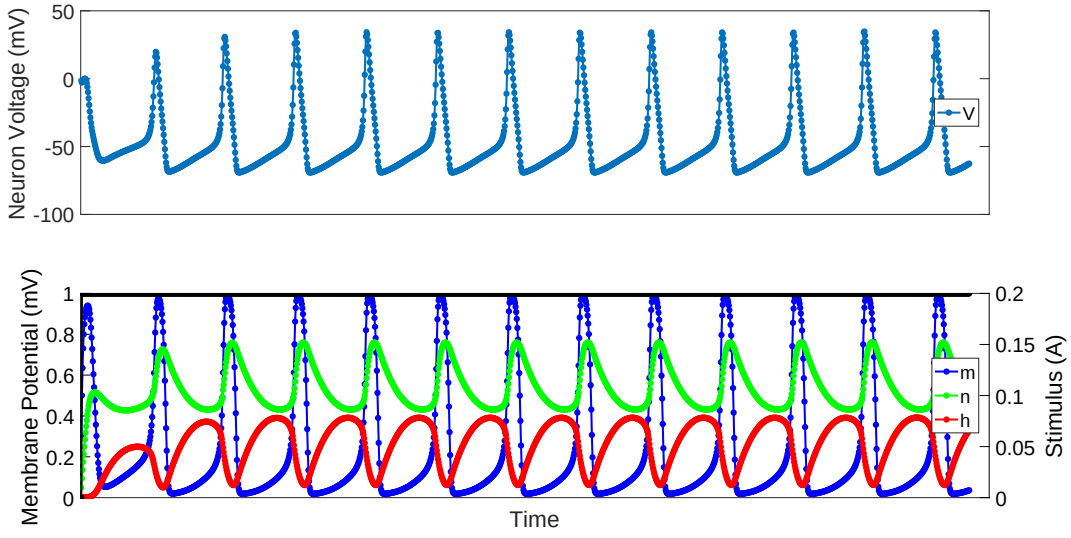


Figure F.5: Hardware implementation of a Hodgkin-Huxley neuron using cubic curve fitting approximation for the exponential function. The neurons response to a step input of 0.2 A (black trace, bottom graph). m, n, and h are the membrane potentials internal to the neuron.

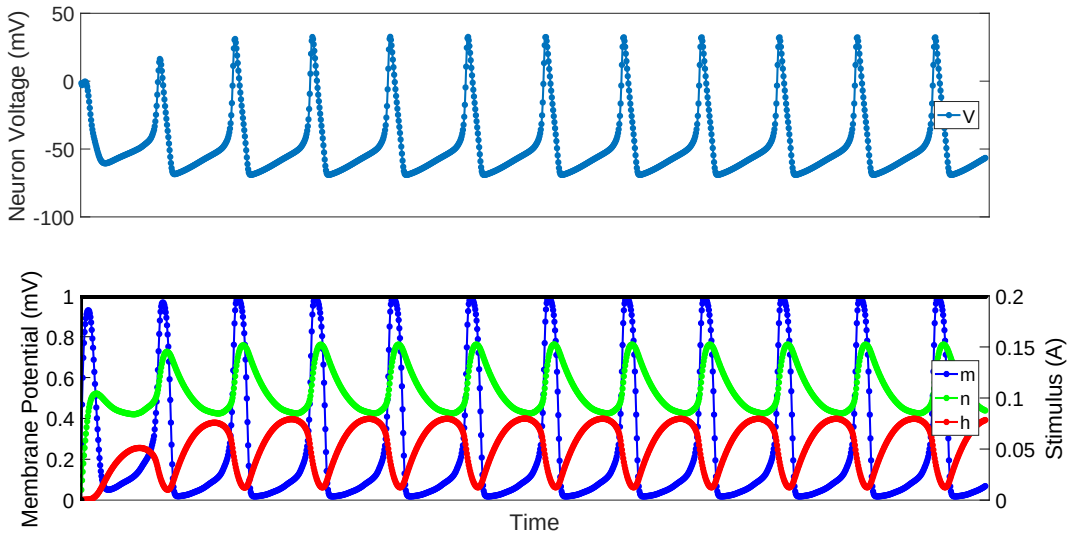


Figure F.6: Hardware implementation of a Hodgkin-Huxley neuron using hybrid single line piecewise linear approximation and $1 + x$ small input approximation for the exponential function. The neurons response to a step input of 0.2 A (black trace, bottom graph). m, n, and h are the membrane potentials internal to the neuron.

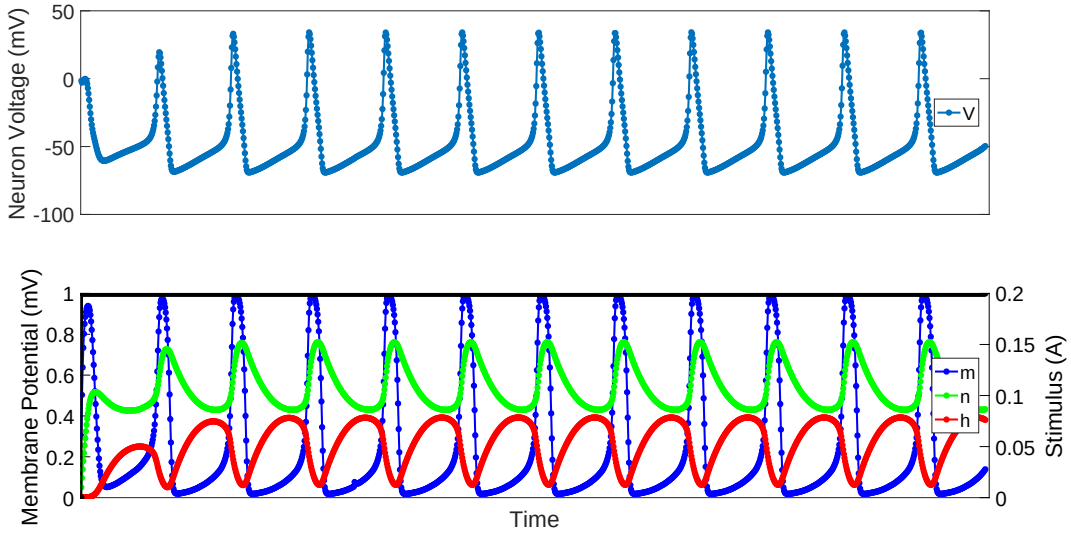


Figure F.7: Hardware implementation of a Hodgkin-Huxley neuron using hybrid two line piecewise linear approximation and $1 + x$ small input approximation for the exponential function. The neurons response to a step input of 0.2 A (black trace, bottom graph). m, n, and h are the membrane potentials internal to the neuron.

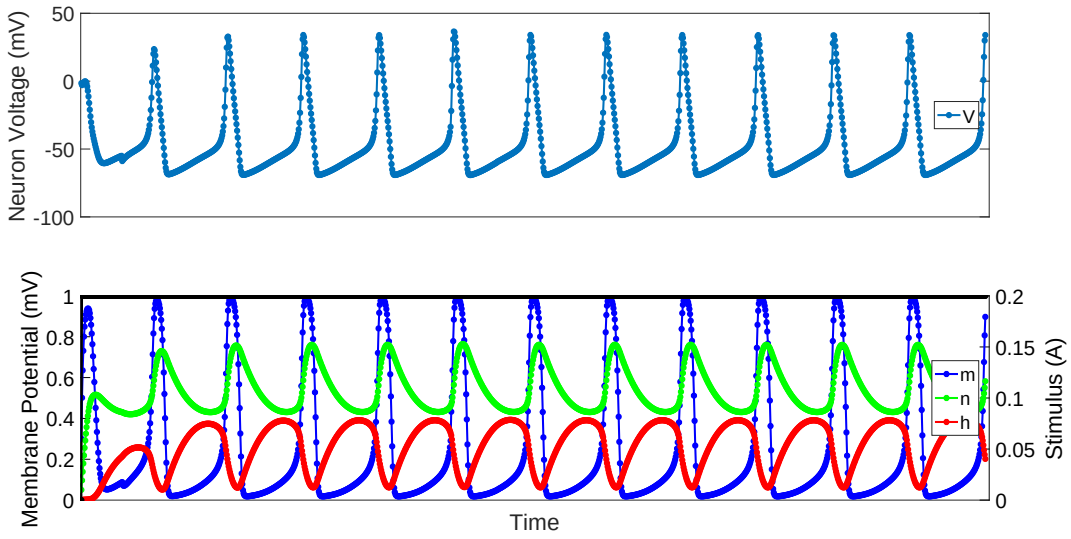


Figure F.8: Hardware implementation of a Hodgkin-Huxley neuron using hyrbid four line piecewise linear approximation and $1 + x$ small input approximation for the exponential function. The neurons response to a step input of 0.2 A (black trace, bottom graph). m, n, and h are the membrane potentials internal to the neuron.

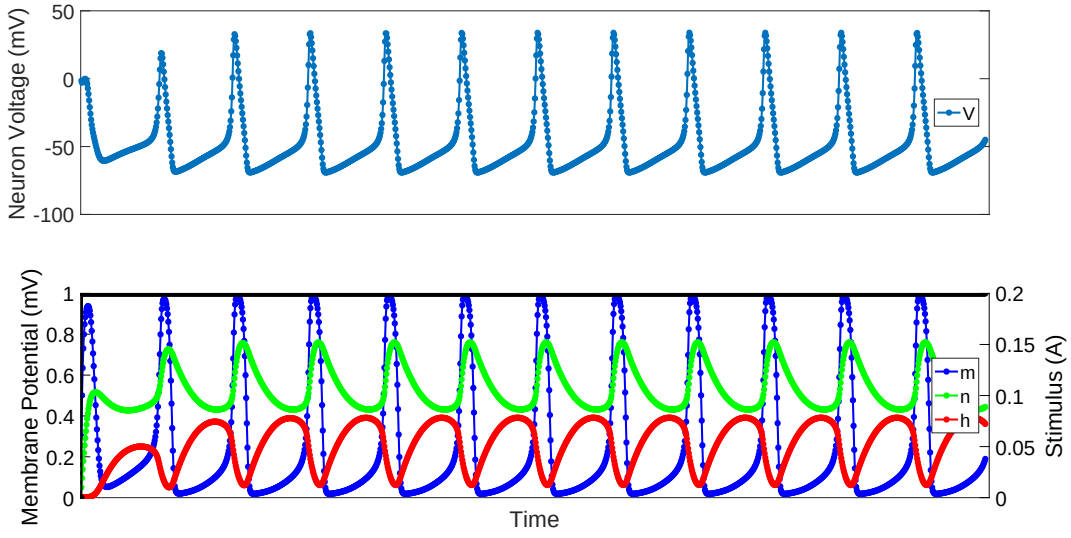


Figure F.9: Hardware implementation of a Hodgkin-Huxley neuron using hybrid quadratic curve approximation and $1 + x$ small input approximation for the exponential function. The neurons response to a step input of 0.2 A (black trace, bottom graph). m, n, and h are the membrane potentials internal to the neuron.

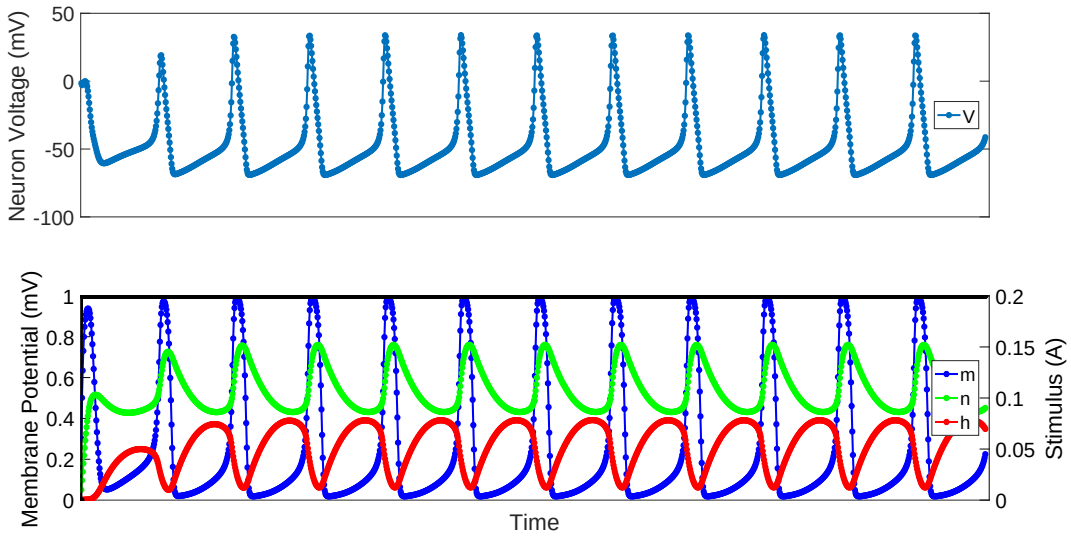


Figure F.10: Hardware implementation of a Hodgkin-Huxley neuron using hybrid cubic curve approximation and $1+x$ small input approximation for the exponential function. The neurons response to a step input of 0.2 A (black trace, bottom graph). m, n, and h are the membrane potentials internal to the neuron.

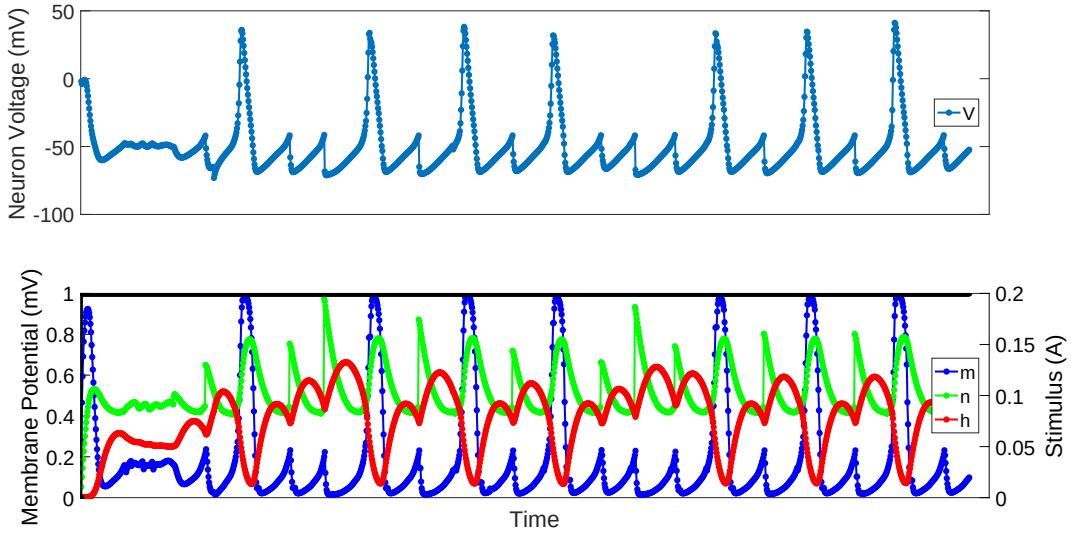


Figure F.11: Hardware implementation of a Hodgkin-Huxley neuron using single line piecewise linear approximation with an integer division step for the exponential function. The neurons response to a step input of 0.2 A (black trace, bottom graph). m, n, and h are the membrane potentials internal to the neuron.

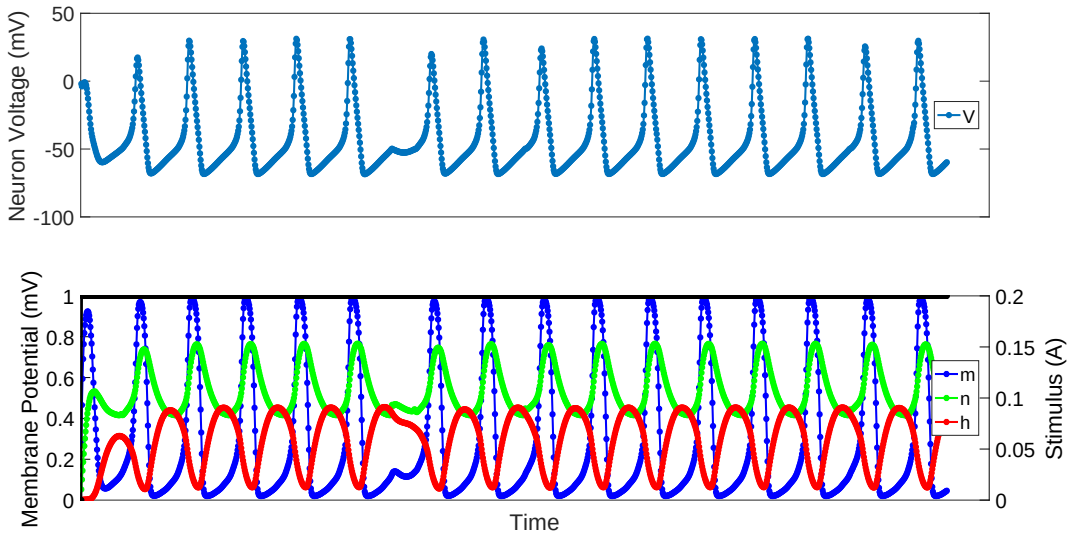


Figure F.12: Hardware implementation of a Hodgkin-Huxley neuron using single line piecewise linear approximation with a floating-point multiplication step for the exponential function. The neurons response to a step input of 0.2 A (black trace, bottom graph). m, n, and h are the membrane potentials internal to the neuron.

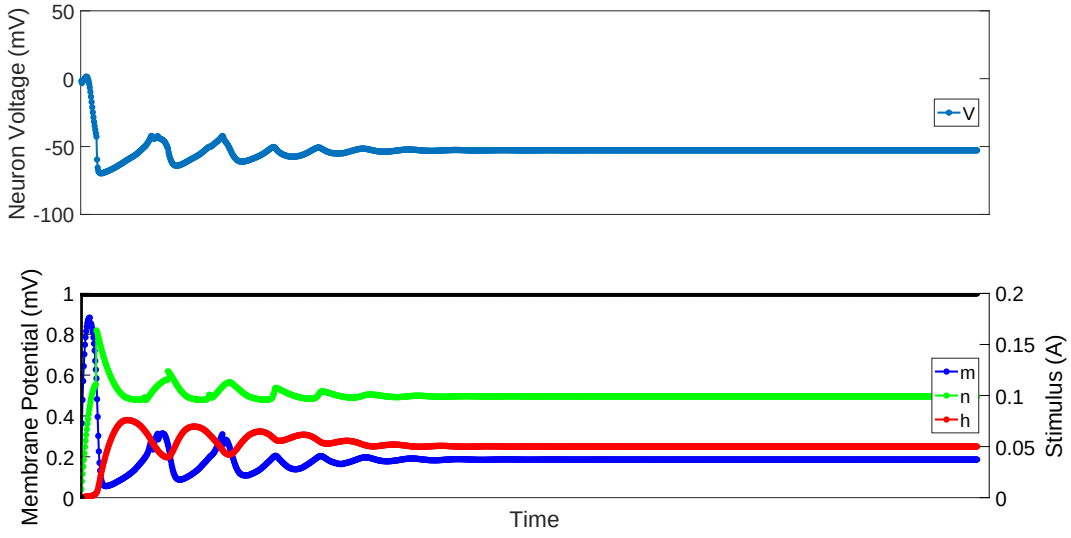


Figure F.13: Hardware implementation of a Hodgkin-Huxley neuron using a 2^x approximation for the exponential function. The neurons response to a step input of 0.2 A (black trace, bottom graph). m, n, and h are the membrane potentials internal to the neuron.

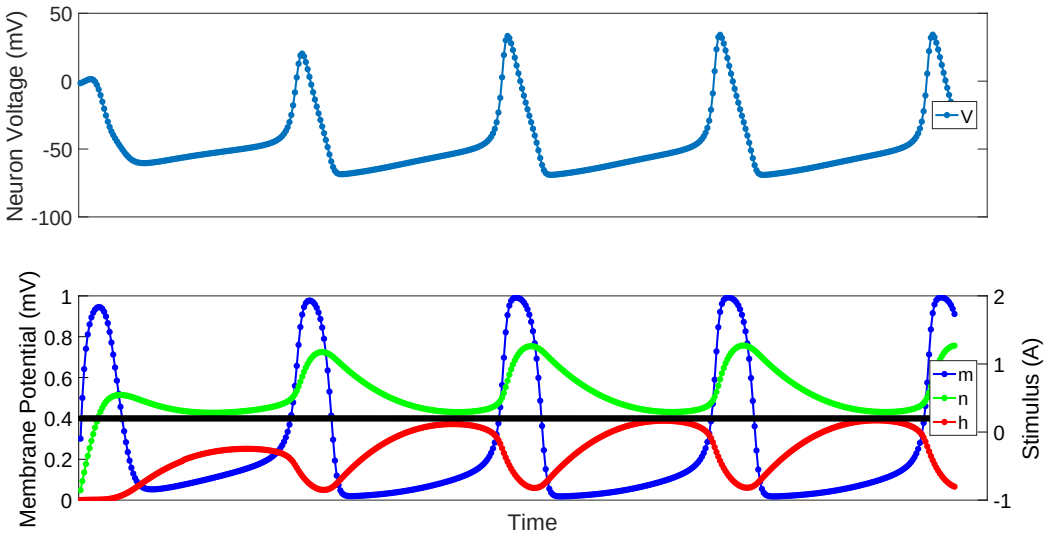


Figure F.14: Hardware implementation of a Hodgkin-Huxley neuron using Euler's approximation for the exponential function. The neurons response to a step input of 0.2 A (black trace, bottom graph). m, n, and h are the membrane potentials internal to the neuron.

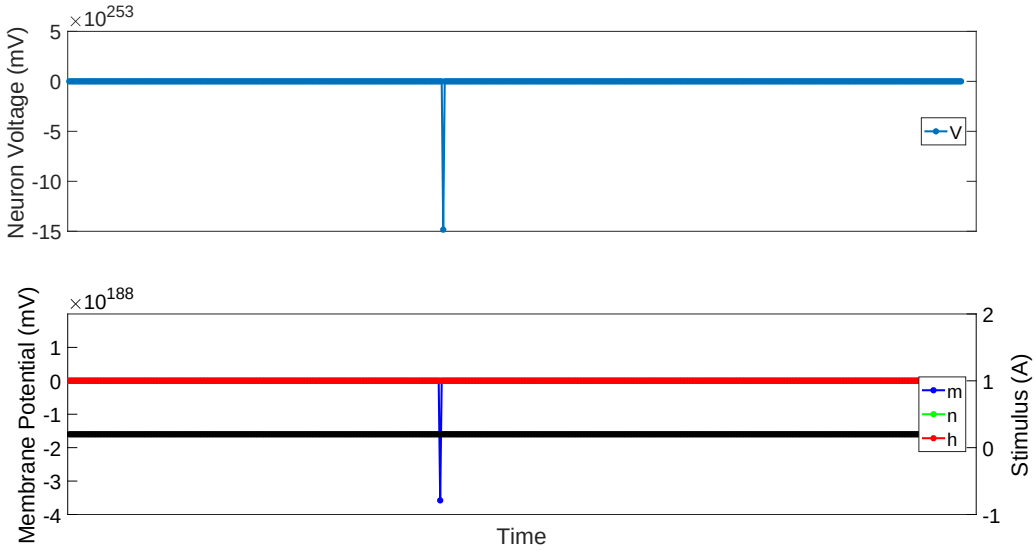


Figure F.15: Hardware implementation of a Hodgkin-Huxley neuron using a power series approximation for the exponential function. The neurons response to a step input of 0.2 A (black trace, bottom graph). m, n, and h are the membrane potentials internal to the neuron.

F.2 Impulse responses

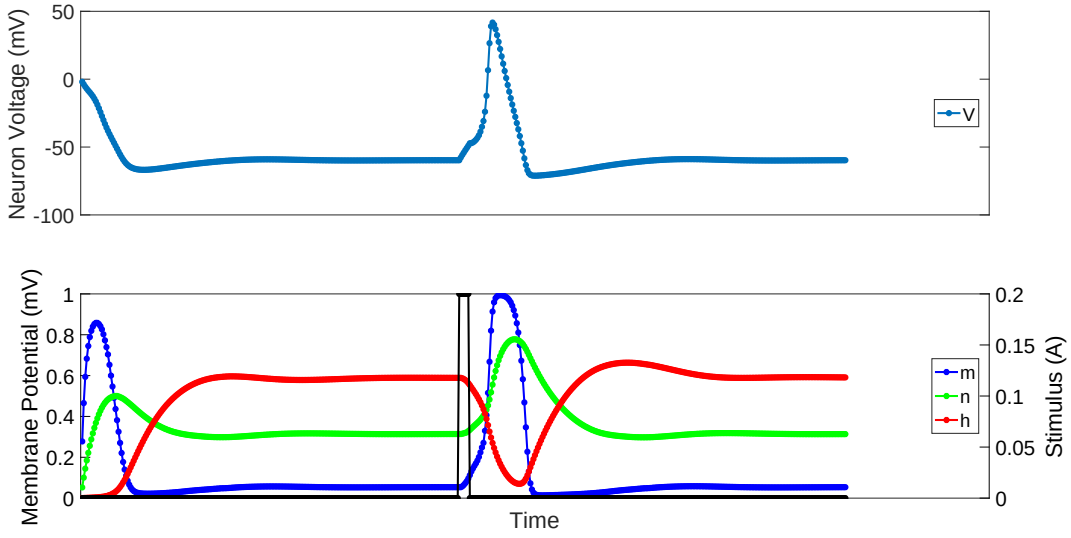


Figure F.16: Hardware implementation of a Hodgkin-Huxley neuron using single line piecewise linear approximation for the exponential function. The neurons response to a impulse of 0.2 A (black trace, bottom graph) lasting 50 μs. m, n, and h are the membrane potentials internal to the neuron.

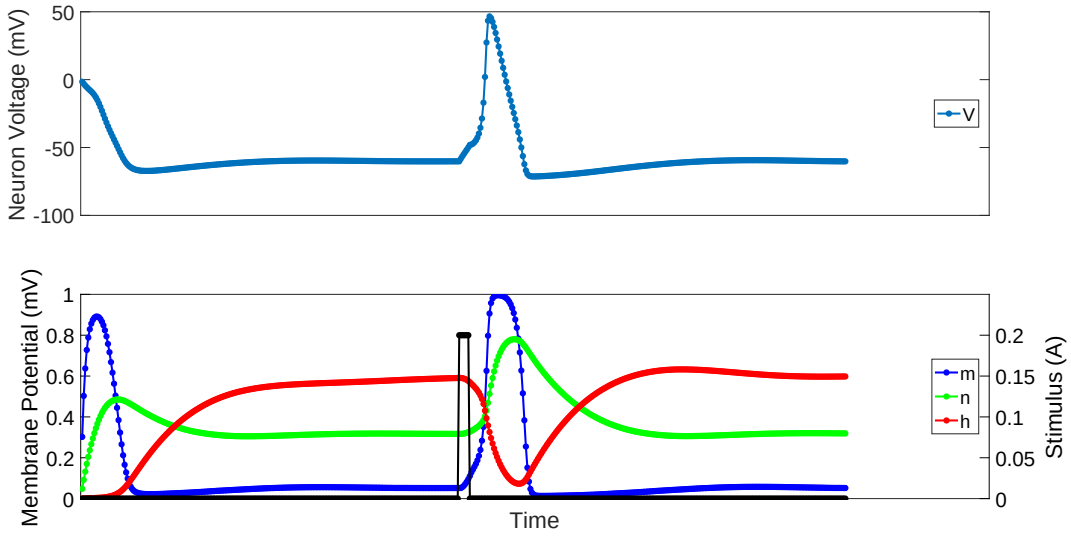


Figure F.17: Hardware implementation of a Hodgkin-Huxley neuron using two line piecewise linear approximation for the exponential function. The neurons response to a impulse of 0.2 A (black trace, bottom graph) lasting 50 μs. m, n, and h are the membrane potentials internal to the neuron.

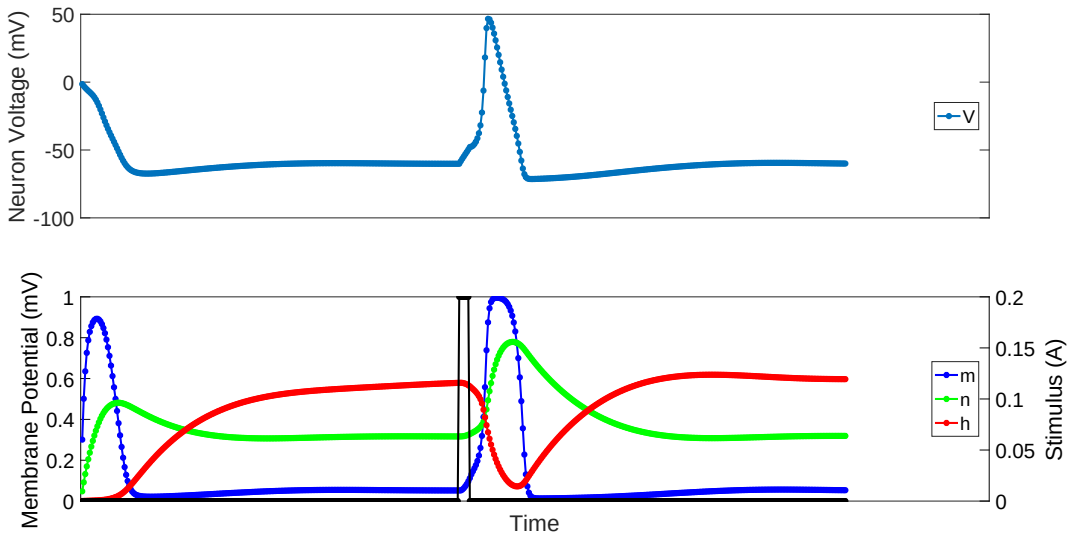


Figure F.18: Hardware implementation of a Hodgkin-Huxley neuron using four line piecewise linear approximation for the exponential function. The neurons response to a impulse of 0.2 A (black trace, bottom graph) lasting 50 μs. m, n, and h are the membrane potentials internal to the neuron.

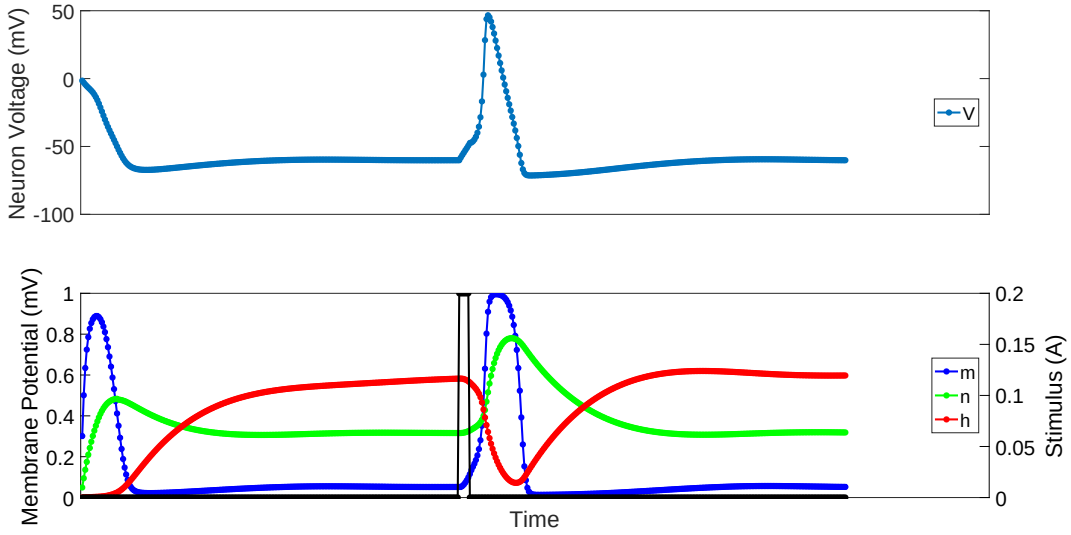


Figure F.19: Hardware implementation of a Hodgkin-Huxley neuron using quadratic curve fitting approximation for the exponential function. The neurons response to a impulse of 0.2 A (black trace, bottom graph) lasting 50 μs. m, n, and h are the membrane potentials internal to the neuron.

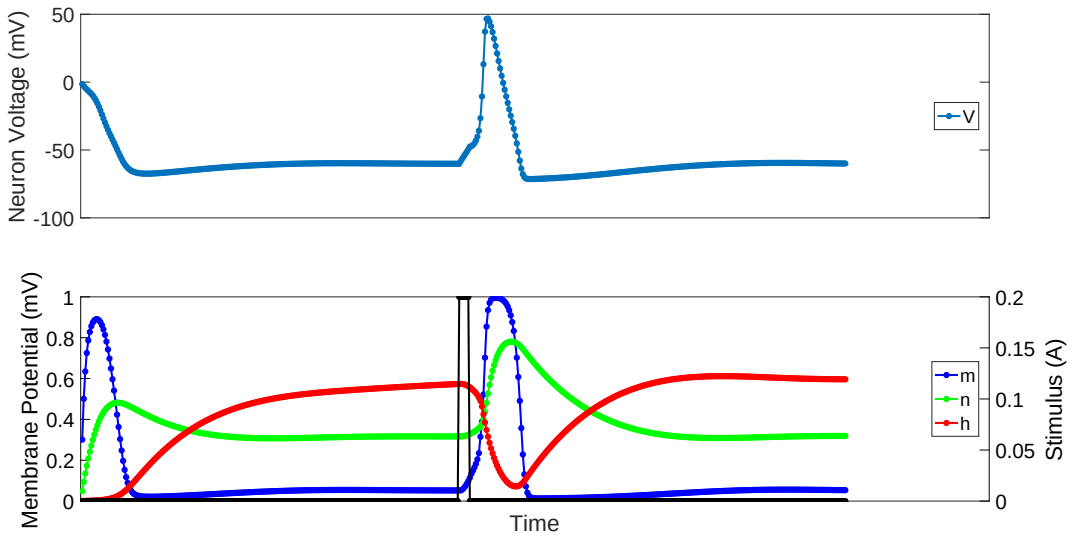


Figure F.20: Hardware implementation of a Hodgkin-Huxley neuron using cubic curve fitting approximation for the exponential function. The neurons response to a impulse of 0.2 A (black trace, bottom graph) lasting 50 μs. m, n, and h are the membrane potentials internal to the neuron.

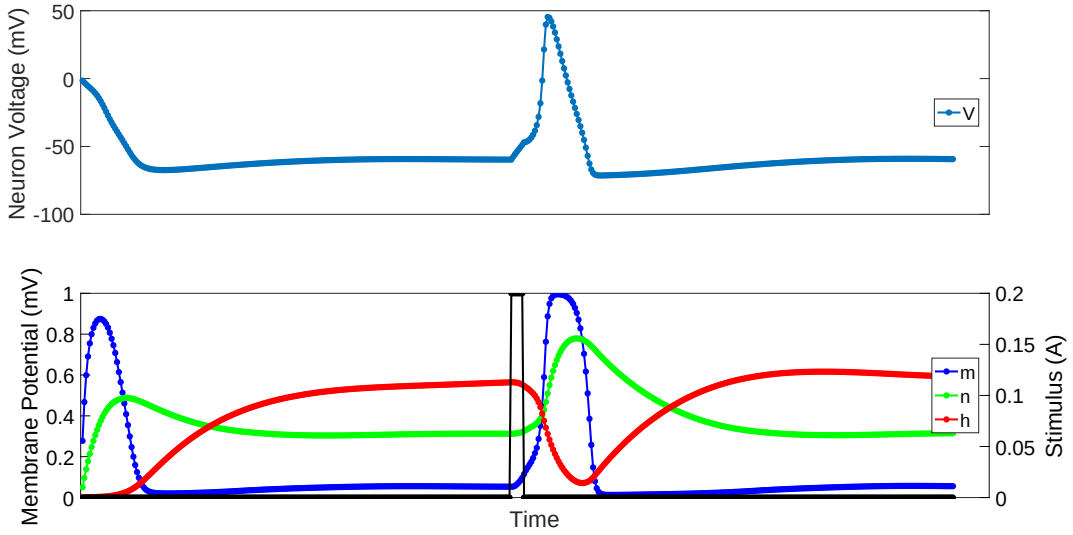


Figure F.21: Hardware implementation of a Hodgkin-Huxley neuron using hybrid single line piecewise linear approximation and $1 + x$ small input approximation for the exponential function. The neurons response to a impulse of 0.2 A (black trace, bottom graph) lasting 50 μs. m, n, and h are the membrane potentials internal to the neuron.

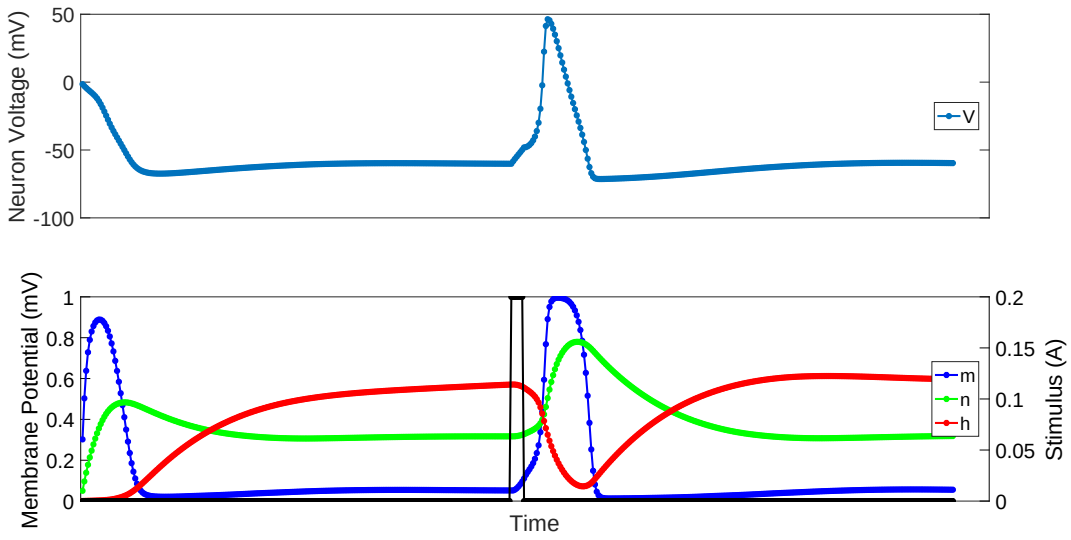


Figure F.22: Hardware implementation of a Hodgkin-Huxley neuron using hybrid two line piecewise linear approximation and $1 + x$ small input approximation for the exponential function. The neurons response to a impulse of 0.2 A (black trace, bottom graph) lasting 50 μs. m, n, and h are the membrane potentials internal to the neuron.

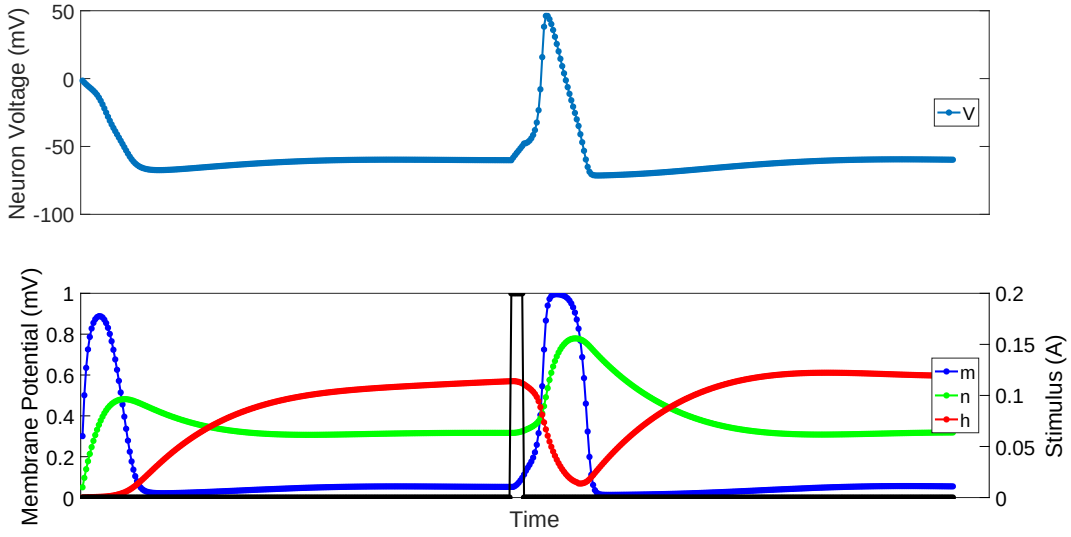


Figure F.23: Hardware implementation of a Hodgkin-Huxley neuron using hyrbid four line piecewise linear approximation and $1 + x$ small input approximation for the exponential function. The neurons response to a impulse of 0.2 A (black trace, bottom graph) lasting 50 μs. m, n, and h are the membrane potentials internal to the neuron.

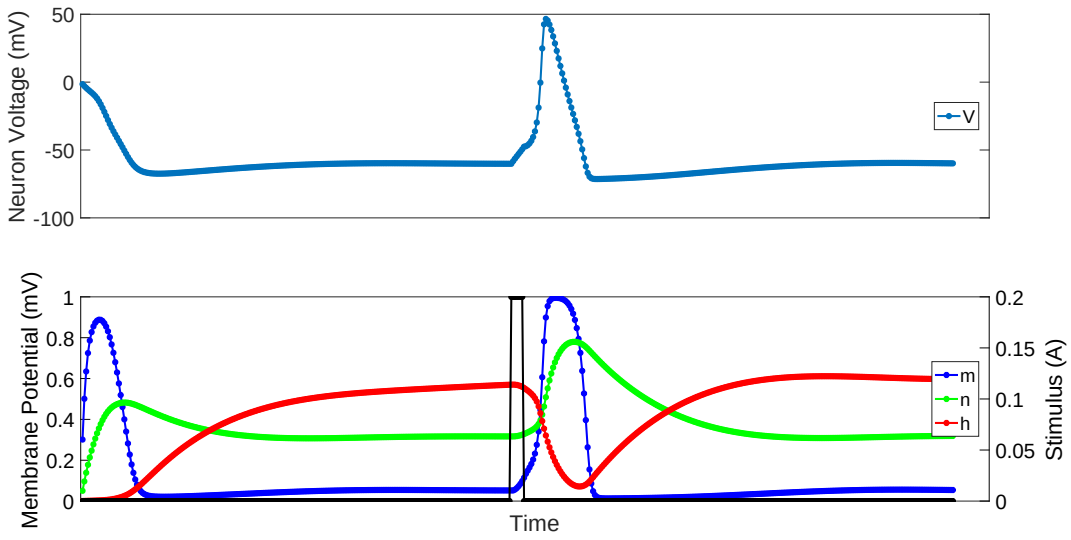


Figure F.24: Hardware implementation of a Hodgkin-Huxley neuron using hybrid quadratic curve approximation and $1 + x$ small input approximation for the exponential function. The neurons response to a impulse of 0.2 A (black trace, bottom graph) lasting 50 μs. m, n, and h are the membrane potentials internal to the neuron.

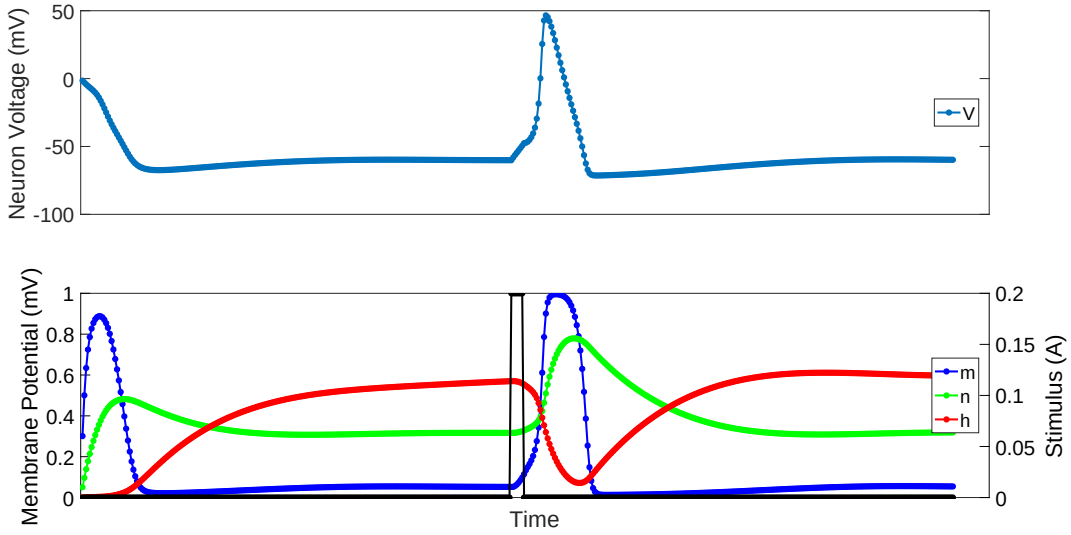


Figure F.25: Hardware implementation of a Hodgkin-Huxley neuron using hybrid cubic curve approximation and $1+x$ small input approximation for the exponential function. The neurons response to a impulse of 0.2 A (black trace, bottom graph) lasting 50 μs. m, n, and h are the membrane potentials internal to the neuron.

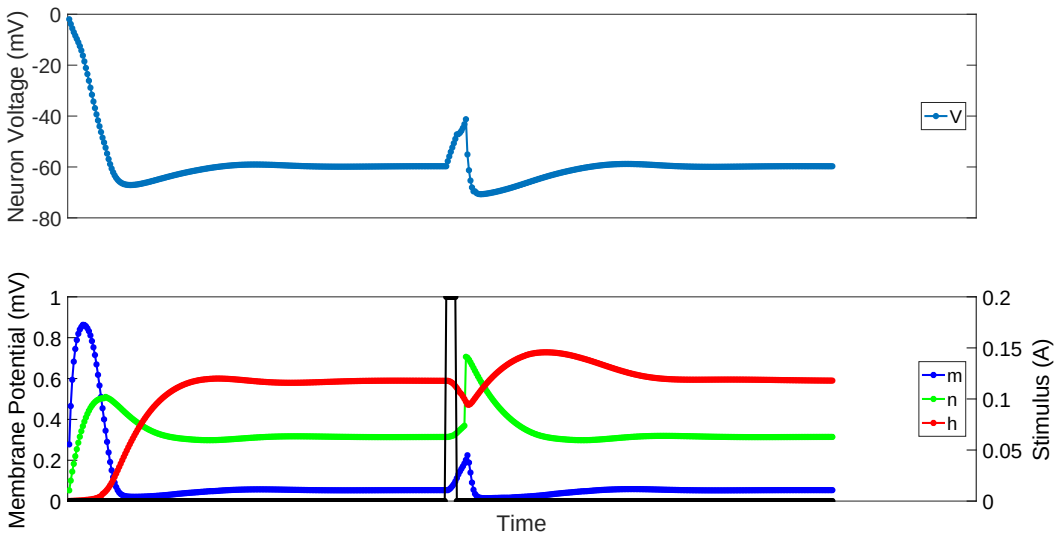


Figure F.26: Hardware implementation of a Hodgkin-Huxley neuron using single line piecewise linear approximation with an integer division step for the exponential function. The neurons response to a impulse of 0.2 A (black trace, bottom graph) lasting 50 μs. m, n, and h are the membrane potentials internal to the neuron.

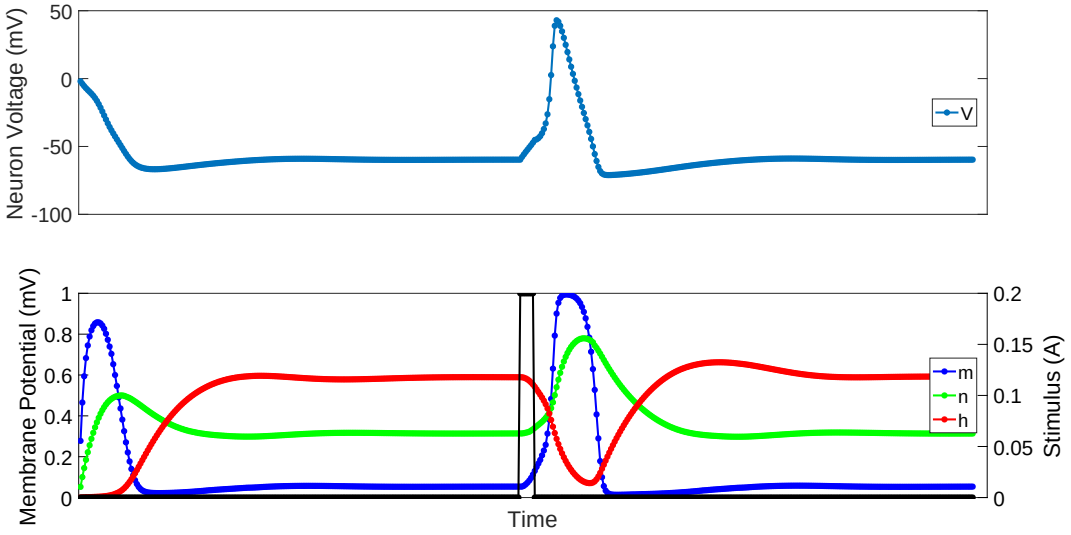


Figure F.27: Hardware implementation of a Hodgkin-Huxley neuron using single line piecewise linear approximation with a floating-point multiplication step for the exponential function.The neurons response to a impulse of 0.2 A (black trace, bottom graph) lasting 50 μs. m, n, and h are the membrane potentials internal to the neuron.

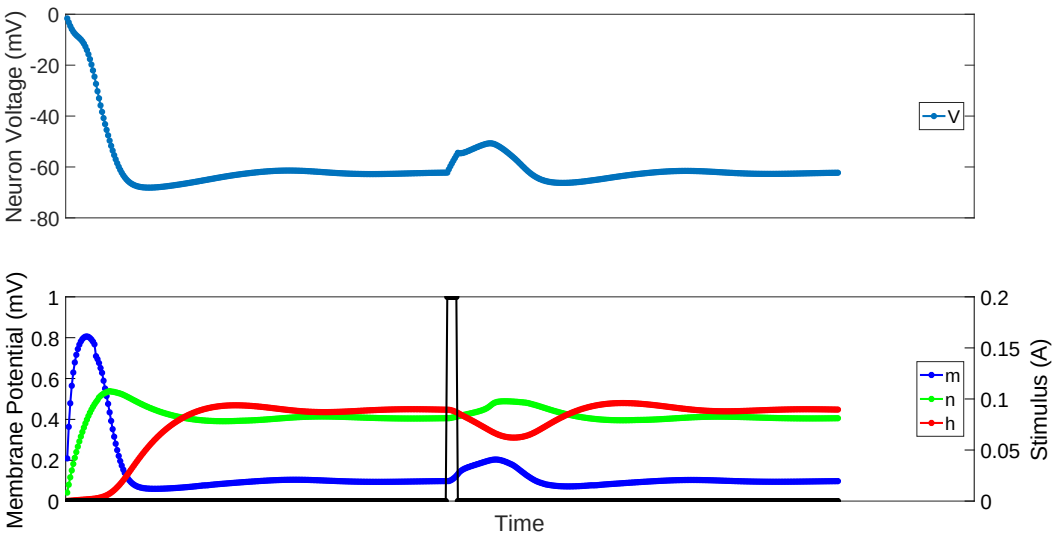


Figure F.28: Hardware implementation of a Hodgkin-Huxley neuron using a 2^x approximation for the exponential function. The neurons response to a impulse of 0.2 A (black trace, bottom graph) lasting 50 μs. m, n, and h are the membrane potentials internal to the neuron.

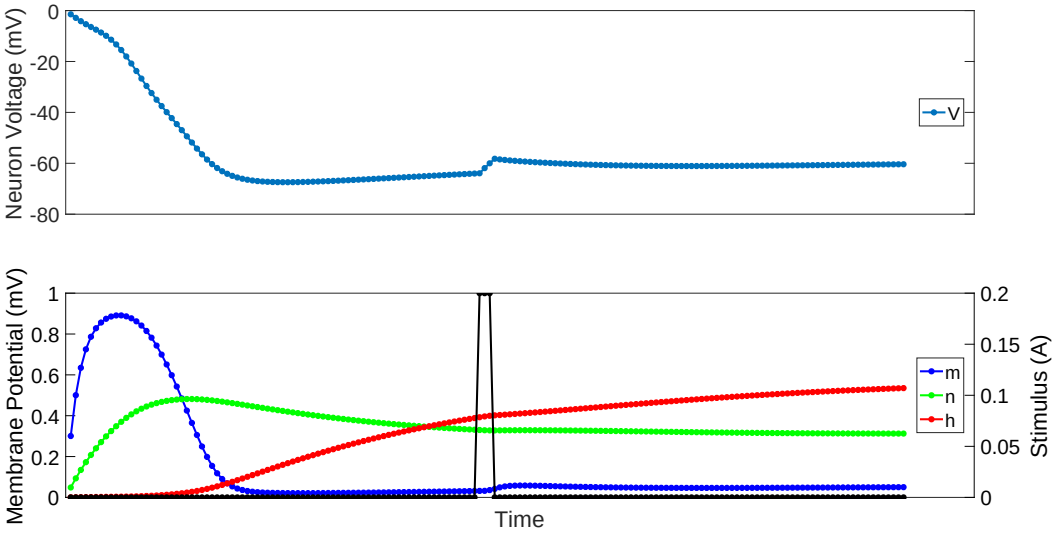


Figure F.29: Hardware implementation of a Hodgkin-Huxley neuron using Euler's approximation for the exponential function.The neurons response to a impulse of 0.2 A (black trace, bottom graph) lasting 50 μs. m, n, and h are the membrane potentials internal to the neuron.

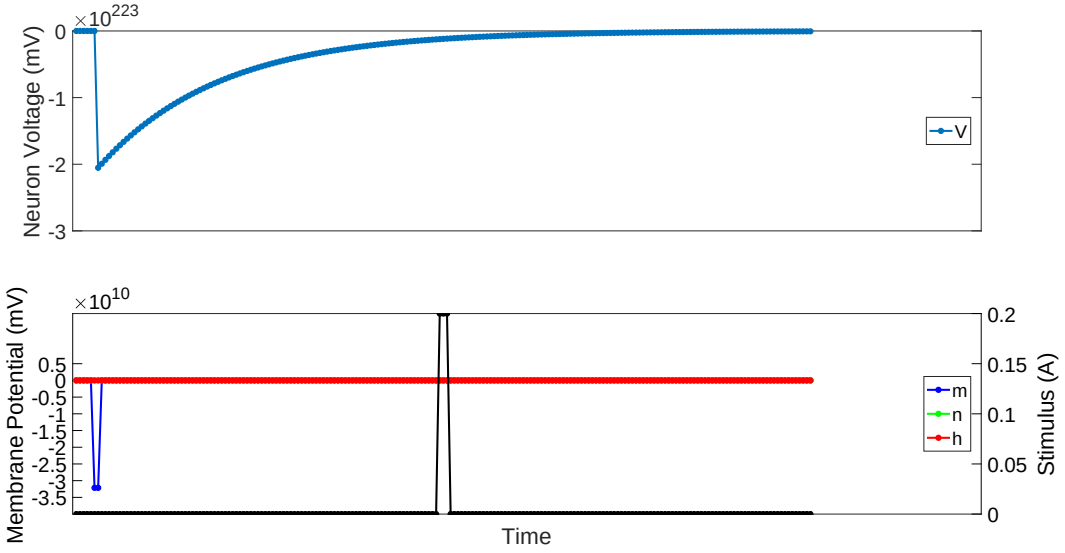


Figure F.30: Hardware implementation of a Hodgkin-Huxley neuron using a power series approximation for the exponential function. The neurons response to a impulse of 0.2 A (black trace, bottom graph) lasting 50 μs. m, n, and h are the membrane potentials internal to the neuron.

F.3 Resource and performance metrics

Table F.1: Resource and performance metrics for FPGA implementations of the Hodgkin-Huxley model of a nuron using different hardware approximators for the exponential function. Implementations are in double floating-point precision and the fitter seed is 1.

Module	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs Recoverable by Dense Packing	[C] Estimate of ALMs Unavailable	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85°C Restricted FMax (MHz)	0°C Restricted FMax (MHz)
single line	128488.6 (11244.7)	194178.5 (12194.7)	66488.4 (950.4)	798.5 (0.4)	156445 (17480)	297656 (64)	352	99.29	96.8
single line pipe	129675.7 (11260.5)	195699.5 (12246.1)	66888.2 (986.1)	864.5 (0.5)	156302 (17489)	301926 (64)	352	109.57	108.27
double line	130210.7 (11274.1)	196758.5 (12162.4)	67466.4 (888.8)	918.5 (0.5)	161463 (17491)	297666 (64)	352	85.74	85.96
double line pipe	131597.4 (11346.1)	198380.2 (12247.0)	67859.8 (901.4)	1077.0 (0.6)	161764 (17496)	301996 (64)	352	105.84	102.69
four line	130295.8 (11241.3)	197115.0 (12133.8)	67789.2 (893.0)	970.0 (0.5)	161953 (17529)	297666 (64)	352	83.04	83.44
four line pipe	131812.4 (11323.5)	198828.2 (12245.9)	68020.3 (922.8)	1004.5 (0.4)	162195 (17501)	302236 (64)	352	105.35	100.87
quad	136385.6 (11241.6)	206629.9 (12174.2)	71186.8 (932.9)	942.5 (0.4)	181351 (17480)	298142 (64)	352	55.49	54.87
quad pipe	141810.7 (11401.4)	213894.3 (12391.2)	73083.1 (991.8)	999.5 (2.0)	193867 (17475)	304943 (64)	352	99.95	101.3
cubic	147903.3 (11210.4)	224752.7 (12179.2)	78414.0 (969.1)	1564.5 (0.3)	218993 (17480)	298223 (64)	352	36.09	35.25
cubic pipe	162429.5 (11430.7)	244593.2 (12300.9)	85138.1 (872.7)	2974.5 (2.5)	254142 (17480)	311003 (64)	352	93.34	95.1
hybrid single line	140860.6 (12820.1)	212048.0 (13864.6)	72214.9 (1046.0)	1027.5 (1.4)	167931 (20119)	329006 (64)	352	93.36	90.53
hybrid single line pipe	141097.3 (12919.8)	212273.0 (14031.6)	72147.2 (1112.5)	971.5 (0.7)	167652 (20122)	329016 (64)	352	112.35	111.62
hybrid double line	142574.6 (12873.3)	215236.5 (13935.4)	73683.4 (1062.5)	1021.5 (0.4)	172916 (20122)	329056 (64)	352	84.98	83.44
hybrid double line pipe	143041.6 (12870.2)	214989.5 (13985.0)	72947.5 (1116.2)	999.5 (1.4)	173245 (20120)	329086 (64)	352	104.42	101.15
hybrid four line	142677.2 (12916.3)	215450.0 (13936.7)	73681.3 (1020.9)	908.5 (0.5)	173576 (20162)	329056 (64)	352	84.05	82.35
hybrid four line pipe	143180.6 (12892.8)	215388.6 (14027.8)	73123.4 (1135.5)	915.5 (0.5)	173605 (20129)	329326 (64)	352	104.87	100.88
hybrid quad	148718.8 (12821.2)	224683.2 (13930.9)	76999.5 (1110.1)	1035.0 (0.4)	192715 (20125)	329492 (64)	352	57.31	57.44
hybrid quad pipe	152677.5 (12992.5)	229721.5 (14095.5)	78027.0 (1104.4)	983.0 (1.4)	205213 (20123)	330613 (64)	352	100.88	102.18
hybrid cubic	160470.3 (12838.2)	242596.9 (13916.5)	84026.7 (1078.6)	1900.0 (0.3)	230349 (20123)	329573 (64)	352	35.5	34.48
hybrid cubic pipe	173782.0 (13017.1)	260793.0 (14017.5)	90105.0 (1002.9)	3094.0 (2.4)	265491 (20108)	337973 (64)	352	92.08	92.54
single line fp mult	115529.7 (11091.8)	174032.3 (12090.2)	58930.2 (999.4)	427.5 (0.9)	110213 (17174)	304713 (64)	352	122.19	120.48
two to the x	110896.2 (11047.6)	167182.5 (12030.2)	56758.3 (983.3)	472.0 (0.6)	101750 (17156)	296383 (64)	320	141.1	134.44

Appendix G

Graphics shaders and rasterization unit resource demand using different floating point precisions

G.1 Vertex shader

Table G.1: Resource requirements and timing analysis for a selection of vertex shaders implementated on an FPGA. Implementations are using double-precision floating-point accuracy.

Module	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs Recoverable by Dense Packing	[C] Estimate of ALMs Unavailable	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
Dedicated Z rotate	9314.1 (0.0)	13701.7 (0.0)	4466.2 (0.0)	78.7 (0.0)	9878 (0)	25008 (0)	32	93.55	90.49
Generic world matrix vector multipler	2643.5 (0.0)	3421.2 (0.0)	806.7 (0.0)	29.0 (0.0)	3182 (0)	6064 (0)	4	97.32	94.2

Table G.2: Resource requirements and timing analysis for a selection of vertex shaders implementated on an FPGA. Implementations are using single-precision floating-point accuracy.

Module	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs Recoverable by Dense Packing	[C] Estimate of ALMs Unavailable	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
Dedicated Z rotate	4820.3 (0.0)	6995.4 (0.0)	2204.6 (0.0)	29.5 (0.0)	5302 (0)	12544 (0)	8	158.03	153.4
Generic world matrix vector multipler	1442.8 (0.0)	1809.6 (0.0)	377.3 (0.0)	10.5 (0.0)	1831 (0)	3194 (0)	1	158.15	153.82
Generic world matrix vector multipler with forward lighting calculations	32390.5 (462.1)	36707.2 (498.2)	4419.4 (47.3)	102.7 (11.1)	23482 (600)	102548 (1067)	18	145.71	147.93

G.2 Fragment shader

Table G.3: Resource requirements and timing analysis for a selection of fragment shaders implementated on an FPGA. Implementations are using double-precision floating-point accuracy.

Module	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs Recoverable by Dense Packing	[C] Estimate of ALMs Unavailable	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
Gradient interpolation shader	8911.9 (387.0)	13927.5 (415.9)	5063.3 (29.4)	47.7 (0.5)	7130 (741)	27073 (774)	16	90.66	87.9
Point light shader using forward lighting calculations	28981.4 (20663.9)	36545.7 (25046.1)	7709.1 (4474.6)	144.8 (92.4)	12194 (2599)	74881 (53325)	52	79.9	83.64

Table G.4: Resource requirements and timing analysis for a selection of fragment shaders implementated on an FPGA. Implementations are using single-precision floating-point accuracy.

Module	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs Recoverable by Dense Packing	[C] Estimate of ALMs Unavailable	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85°C Restricted f_{max} (MHz)	0°C Restricted f_{max} (MHz)
Flat colour shader	85.5 (85.5)	104.5 (104.5)	19.0 (19.0)	0.0 (0.0)	115 (115)	226 (226)	0	614.25	660.5
Gradient interpolation shader	4708.3 (275.8)	7150.1 (290.6)	2462.7 (14.7)	21.0 (0.0)	3961 (517)	13831 (550)	4	133.71	130.77
Point light shader using forward lighting calculations	14892.8 (10715.6)	19231.4 (13126.6)	4456.1 (2506.5)	117.5 (95.4)	6599 (1797)	37704 (26797)	13	109.99	112.88

Appendix H

FPGA based implementation of a GPU using different floating point precisions

Table H.1: Resource requirements and timing analysis for a variety of full graphics processors implemented on an FPGA using the individual components listed earlier. Implementations are using single-precision floating-point accuracy.

Pipeline Type	ALMs Needed [=A-B+C]	[A] ALMs used in Final Placement	[B] Estimate of ALMs Recoverable by Dense Packing	[C] Estimate of ALMs Unavailable	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85^{circ}C Restricted FMax (MHz)	0^{circ}C Restricted FMax (MHz)
Flat colour fragment shader with generic matrix/vector multiplier vertex shader	26079.5 (167.0)	35322.0 (298.8)	9484.0 (131.8)	241.5 (0.0)	29655 (15)	61209 (643)	13	48.74	51.29
Gradient colour fragment shader with generic matrix/vector multiplier vertex shader	30982.0 (172.5)	36060.5 (235.5)	5284.5 (63.0)	206.0 (0.1)	33936 (25)	74695 (643)	17	45.79	47.94